(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 540 151 A2**

(12) **EUROPEAN PATENT APPLICATION**

(21) Application number: **92308137.6**

(51) Int. Cl.5: **G06F 9/46**

(22) Date of filing: **08.09.92**

(30) Priority: **31.10.91 IL 99923**

(43) Date of publication of application:
05.05.93 Bulletin 93/18

(84) Designated Contracting States:
DE FR GB

(71) Applicant: **International Business Machines Corporation**
Old Orchard Road
Armonk, N.Y. 10504(US)

(72) Inventor: **Allon, David**
49/8 Meir Nakar Street
Jerusalem(IL)

Inventor: **Bach, Moshe**
Trumpeldor Street 5a
Haifa(IL)
Inventor: **Moatti, Yosef**
68/56 Hanita Street
Haifa(IL)
Inventor: **Teperman, Abraham**
46 Haviva Reich Street
Haifa(IL)

(74) Representative: **Blakemore, Frederick Norman**
IBM United Kingdom Limited Intellectual Property Department Hursley Park
Winchester Hampshire SO21 2JN (GB)

(54) **Method of operating a computer in a network.**

(57) A method is described of operating a computer in a network of computers using an improved load balancing technique, the method comprising: generating logical links between the computer and other computers in the network so that a tree structure is formed, the computer being logically linked to one computer higher up the tree and a number of computers lower down the tree; maintaining in the computer stored information regarding the current load on the computer and the load on at least some of the other computers in the network by causing the computer periodically to distribute the information to the computers to which it is logically linked, to receive from said computers similar such information and to update its own information in accordance therewith, so that the information can be used to determine a computer in the network that can accept extra load. A sender-initiated embodiment of the invention includes the further step of, when the computer is overloaded, using the information to determine a computer that can accept extra load and transferring at least one task to that computer. The load balancing technique is scalable, fault tolerant, flexible and supports clustering thus making it suitable for use in networks having very large numbers of computers.

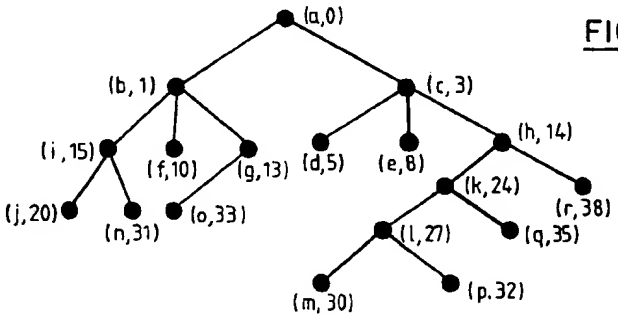

FIG. 3a

EP 0 540 151 A2

The invention relates to methods of operating computers in networks and to computers capable of operating in networks.

Local area networks of computers are an indispensable resource in many organisations. One problem in a large network of computers is that it is sometimes difficult to make efficient use of all the computers in the network. Load sharing or balancing techniques increase system throughput by attempting to keep all computers busy. This is done by off-loading processes from overloaded computers to idle ones thereby equalising load on all machines and minimising overall response time.

Load balancing methods can be classified according to the method used to achieve balancing. They can be 'static' or 'dynamic' and central' or 'distributed'.

In static load balancing, a fixed policy – deterministic or probabilistic – is followed independent of the current system state. Static load balancing is simple to implement and easy to analyse with queuing models. However, its potential benefit is limited since it does not take into account changes in the global system state. For example, a computer may migrate tasks to a computer which is already overloaded.

In dynamic schemes the system notes changes in its global status and decides whether to migrate tasks based on the current state of the computers. Dynamic policies are inherently more complicated than static policies since they require each computer to have knowledge of the state of other computers in the system. This information must be continuously updated.

In a central scheme one computer contains the global state information and makes all the decisions. In a distributed scheme no one computer contains global information. Each computer makes migration decisions based on the state information it has.

Load-balancing methods can be further classified as 'sender-initiated' and 'receiver-initiated'. In sender-initiated policies an overloaded computer seeks a lightly loaded computer. In receiver-initiated policies lightly loaded computers advertise their capability to receive tasks. It has been shown that if costs of task-transfer are comparable under both policies then sender-initiated strategies outperform receiver-initiated strategies for light to moderate system loads, while receiver-initiated schemes are preferable at high system loads.

Conventionally, a dynamic load balancing mechanism is composed of the following three phases:

1. Measuring the load of the local machine.
2. Exchanging local load information with other machines in the network.
3. Transferring a process to a selected machine.

Local load can be computed as a function of the number of jobs on the run queue, memory usage, paging rate, file use and I/O rate, or other resource usage. The length of the run queue is a generally accepted load metric.

The receipt of load information from other nodes gives a snapshot of the system load. Having this information, each computer executes a transfer policy to decide whether to run a task locally or to transfer it to another node. If the decision is to transfer a task then a location policy is executed to determine the node the task should be transferred to.

If a load balancing method is to be effective in networks having large numbers of computers it must be scalable, in the sense that network traffic increases linearly with the size of the network, flexible so that computers can be easily added to or removed from the network, robust in the event of failure of one or more of the computers and must support to some degree clustering of the computers.

Centralised methods, though attractive due to their simplicity are not fault-tolerant. The central computer can detect dead nodes and it takes one or two messages to re-establish connection. However, if the central node fails, the whole scheme falls apart. Load balancing configuration can be re-established by maintaining a prioritised list of alternative administrators in each node or by implementing an election method to elect a new centralised node. When a central node fails, other nodes switch to the new central node. This enhancement, however, increases the management complexity and cost of the method.

In addition, management overhead for a centralised method is unacceptably large. As the number of nodes increases, the time spent by the central node in handling load information increases, and there must come a point at which it will overload.

Several prior art load balancing methods are both dynamic and distributed.

For example, in the method described in Barak A. and Shiloh A. SOFTWARE – PRACTICE AND EXPERIENCE, 15(9):901 – 913, September 1985 [R1] each computer maintains a fixed size load vector that is periodically distributed between computers. Each computer executes the following method. First, the computer updates its own load value. Then, it chooses a computer at random and sends the first half of its load vector to the chosen computer. When receiving a load vector, each computer merges the information with its local load vector according to a predefined rule.

A problem with this method is that the size of the load vector has to be chosen carefully. Finding the optimal value for the load vector is difficult and it must be tuned to a given system. A large vector contains load information for many nodes. This, many nodes will know of a lightly loaded node, increasing the chance that it will quickly receive many migrating processes and will overload. On the other hand, the size of the load vector should not be so small that load values do not propagate through the network in a reasonable time. For large number n of nodes in the network, the expected time to propagate load information through the network is O(log n).

The method described in R1 can handle any number of computers after tuning the length of the load vector. Therefore, the method is scalable. However, it is flexible only up to a point. Adding a small number of nodes to the network does not require any change. Adding a large number of nodes requires changing the size of the load vector in all computers, thus increasing the administrative overhead. The method is fault-tolerant and continues to work in spite of single failures, however there is no built-in mechanism for detecting dead nodes and updating information on them in the load vectors. This information about failed nodes is not propagated and other nodes may continue to migrate processes to them, with the result of decreased response time.

The number of communications per unit time is O(n). However, the method does not support clustering. When a node wants to off-load processes to another node it chooses a candidate from its load vector. Since the information on nodes in the load vector of node p is updated from random nodes, the set of candidates for off-loading is a random subset of n and not a controlled set for p.

In Lin F. C. H. and Keller R. M. PROCEEDINGS OF THE 6TH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 329-336, May 1986 [R2], a distributed and synchronous load balancing method using a gradient model is disclosed. Computers are logically arranged in a grid and the load on the computers is represented as a surface. Each computer interacts solely with its immediate neighbours on the grid. A computer with high load is a 'hill' and an idle computer is a 'valley'. A neutral computer is one which is neither overloaded nor idle. Load balancing is a form of relaxation of the surface. Tasks migrate from hills towards valleys and the surface is flattened. Each computer computes its distance to the nearest idle computer and uses that distance for task migration.

An overloaded node transfers a task to its immediate neighbour in the direction of an idle node. A task continues to move until it reaches an idle node. If a former idle node is overloaded when a task arrives, the task moves to another idle node.

This method, though scalable, is only partially flexible. It is easy to add or remove nodes at the edge of the grid, but it is difficult to do so in the middle since the grid is fixed. To do this, reconfiguration is required, which increases the overhead inherent in the method. Detection of dead nodes is not easy. Since only changes of the state of a node are sent to its neighbours, a node's failure remains undetected until a job is transferred to it. Late detection delays migrating processes and, therefore, increases overall response time. In this method clustering is not supported. Each node transfers processes to one of its immediate neighbours which may transfer it in a different direction. The overloaded node has no control as to where the off-loaded processes will eventually execute. Management overhead to start the grid is low and the number of messages is O(n). Administrative overhead for reconfiguration is high since it requires changes in all neighbouring nodes for a failed node. In this method node overhead is high and network traffic increases because tasks move in hops rather than directly to an idle node and it takes long time to migrate a process.

The buddy set algorithm proposed in Shin K. G. and Chang Y. C. IEEE TRANSACTIONS ON COMPUTERS, 38(8), August 1989 [R3], aims for very fast load sharing. For each node two lists are defined: Its buddy set, the set of its neighbours, and its preferred list, an ordered list of nodes in its buddy set to which tasks are transferred. Each node can be in one of the following states: under-loaded, medium-loaded and overloaded. Each node contains the status of all nodes in its buddy set. Whenever the status of a node changes, it broadcasts its new state to all nodes in its buddy set. The internal order of preferred lists varies per node in order to minimise the probability that two nodes in a buddy set will transfer a task to the same node. When a node is overloaded it scans its preferred list for an under-loaded node and transfers a task to that node. Overloaded nodes drop out of preferred lists. Buddy sets and their preferred lists overlap, so processes migrate around the network.

Again, this method, though scalable in that the number of messages increases linearly with the number of computers in the network n, is not flexible. Adding or removing nodes from the network requires recomputation of the preferred lists in all nodes of the related buddy set. If a node is added to more than one buddy set the recomputation must be done for each node in each buddy set. Detection of dead nodes is difficult for the same reason as in the gradient model described in R2. Clustering is supported but reconfiguration is expensive since it requires recomputation of new buddy sets and preferred lists, as for adding and removing nodes. The administrative overhead inherent in the method is therefore high.

According to a first aspect of the present invention there is provided a method of operating a computer in a network of computers, the method comprising: generating logical links between the computer and other computers in the network so that a tree structure is formed, the computer being logically linked to one computer higher up the tree and a number of computers lower down the tree; and maintaining in the computer stored information regarding the current load on the computer and the load on at least some of the other computers in the network by causing the computer periodically to distribute the information to the computers to which it is linked, to receive from said computers similar such information and to update its own information in accordance therewith, so that the information can be used to determine computers in the network that can accept extra load.

The invention is applicable to both sender-initiated and receiver-initiated load balancing. A embodiment of the invention using a sender-initiated load balancing technique includes the further step of, when the computer is overloaded, using the information to determine a computer that can accept extra load and transferring at least one task to that computer.

There is further provided a method of operating a network of computers by operating each computer using the above method.

The generation of the logical links can be achieved by assigning a rank to each computer, no two computers being assigned the same rank, each computer being logically linked to one computer of lower rank and a number of computers of higher rank to form the tree structure.

The tree structure can be maintained if a computer fails, or is otherwise inoperative, by generating new logical links between each of the computers lower down the tree to which the failed computer was linked and other computers, which have capacity for accepting new downward links.

An improved load balancing technique is employed which has the property of scalability by limiting the communication load on each computer and the knowledge of the network topology which is required to be stored in each computer. The actions taken by each computer are thus independent of the total number of computers in the network.

Since the tree structure is dynamically built and maintained, the method is also flexible. Changing a tree configuration requires simply changing the ranking of the reconfigured computers. The new ranking files have to be consistent with existing ranking files to avoid cycles. A computer is reconfigured by logically disconnecting it so that it appears inoperative to its neighbours, replacing its configuration file, and letting it reconnect to the tree. A new configuration is achieved without disturbing other nodes on the tree and without disrupting local services.

Detection and reconnection of failed nodes is also provided for. If a computer higher up the tree does not respond within a certain time each higher ranked computer previously connected to it tries to relink itself elsewhere in the tree. If it temporarily fails to do this, it acts as root of a disconnected sub-tree, and load balancing occurs within the sub-tree. Eventually, the flags marking nodes as already tried or dead will be reset, and the computer, which periodically tries to attach itself to the tree, will reconnect to the tree.

Advantageously, the periodic distribution of load information across the links of the tree is used by each computer to determine whether or not the computers to which it is linked are operational, each computer seeking, if the computer to which it is linked higher in the tree is not operational, to generate a new link with a computer having lower rank and having the capacity for new downward links and being marked, if one of the computers lower in the tree to which it is linked is not operational, as having capacity to accept new downward links.

In a preferred form of the invention, the information stored in each computer contains a number of entries, each entry containing information regarding the load on a particular one of the computers in the network, i, the number of links in the tree separating it from the computer in which the information is stored, and the name (ie, rank) of the computer, logically linked to the computer in which the information is stored, from which the entry was last received; and wherein when each computer receives the similar information from a computer to which it is logically linked the following steps are performed.

a) the number of links separation value in each entry of the received information is incremented by one;

b) entries in the received information which originated from the receiving computer are deleted;

c) entries in the information already stored in the receiving computer which were received from the sending computer are deleted;

d) the received information is merged with the information already stored in the receiving computer;

e) the merged information is sorted in ascending order of load, entries with equal load being sorted in ascending order of number of links separation from the receiving computer.

Before each distribution of information between computers, entries in the information relating to computers which have the same load and number of links separation from the computer can be randomly permuted, so that the probability that the same entry will appear first in the information stored in different

computers is reduced. Also, if an entry corresponding to the computer appears first in its own sorted information a counter can be attached to the entry and initialised to the negative value of the spare load capacity of the computer, the counter being incremented, if that entry is still first, whenever the information is distributed, and if the counter becomes positive that entry is removed from the information. The probability that the same entry will appear first in the information stored in different computers is thereby reduced.

In an advantageous form of the invention the period of the periodic distribution of the information to a computer higher up the tree is related to the rate at which new tasks are created in the computer and/or the rate at which new tasks are created in the computers to which it is logically linked in the tree structure. This has the advantage that if one or more nodes suddenly become highly loaded this information will spread more quickly throughout the network.

A further advantage of the method provided in the present invention is that the case where the overall load in the network increases steadily is handled gracefully. In such a case, when local load vectors are searched for a target computer, fewer candidates will be found and the number of migrations will decrease. When overall load decreases, more candidates will be found in the local load vectors and migrations will resume. In other words the network performance degrades gracefully during network load saturation and does not crash, rather normal operation is resumed when overall load returns to normal.

Viewed from another aspect the invention enables to be provided a computer capable of operating in a network of similar such computers, the computer comprising: means identifying computers in the network to which the computer is logically linked in a tree structure; means for storing information regarding the load on the computer and at least some of the other computers in the network; means for sending said information to the computers to which the computer is logically linked in the tree structure; means for receiving similar information from said computers to which the computer is logically linked and updating the stored information in accordance therewith, means for selecting one of the other computers in the network having spare load capacity using said information and transferring tasks to the selected computer.

The computer preferably also includes means for accessing a stored list of all the computers in the network; means for selecting a computer from this list as a candidate neighbouring computer for linking in the tree structure; means for sending a message to the selected computer indicating a link request; means for establishing a logical link with the selected computer by updating the identifying means if a positive response from the selected computer is received; means for receiving messages from other computers indicating that a link is requested; means for determining, on receipt of such a link request message, whether or not there is capacity for establishing a downward link and sending a positive response to the sender of the link request message if there is such capacity and updating the identifying means accordingly.

There is also provided a network of such computers.

An embodiment of the invention will now be described, by way of example only, with reference to the accompanying drawing, wherein:

Figures 1a, 1b, 1c and 1d are flow diagrams showing the request and receive phases of the tree generation;

Figures 2a, b and c show stages of the tree generation according to the present invention;

Figures 3a, b and c illustrate an example of tree maintenance in the embodiment of the invention;

Figures 4a, b and c illustrate the handling of clustering of nodes in the embodiment of the invention.

This embodiment of the invention is composed of two main parts. The first part is tree building and maintenance, and the second part is the exchange and maintenance of load balancing information.

TREE BUILDING AND MAINTENANCE

Each computer in the network is assigned a unique rank which is used in building the tree. In the following the term "parent" will be used to refer to a computer of lower rank to which a computer can form an upward link and the term "child" will be used to refer to a computer of higher rank to which a computer can form a downward link.

Each computer has stored in it a configuration file that contains

(i) the maximum number of direct descendants the computer can accept. These are referred to as children slots.

(ii) a list of computers and their respective ranks.

(iii) an ordered list of "favoured parent" nodes, FP. Favoured parents have lower rank than the current computer. The FP list is used by the computer to choose a parent node. The FP list is optional. It can be empty in which case a parent is chosen at random. The FP list includes a subset of all computers of

lower rank.

(iv) the time to wait for parent and children responses.

(v) the time to wait to probe if a node is dead or alive.

A list of candidate parents for a computer CR, which will be referred to as the candidates range of P, limits the range of possible parents. Initially CR of each computer contains all the computers with rank less than the rank of the computer. A node is chosen as a parent only if it is in CR and in FP. If FP is empty or exhausted then only membership of CR is checked.

Each computer performs the following steps to build the tree. There is a request phase in which a computer chooses another computer and requests it to become a parent, and a receive phase in which the computer waits for a reply and responds to requests from other computers.

Request Phase

1. Each computer i picks another computer from its list of favoured parents FP_i. If FP_i is empty or all candidates in FP_i failed, a computer with lesser rank is chosen at random. If the chosen computer j is not in CR_i then the next one in FP_i is chosen or another is randomly picked. If all nodes were tried unsuccessfully then the computer enters the receive phase.
2. i sends the chosen computer j a request_parent message and waits for messages from other computers.
3. If the candidate parent does not respond within a certain period of time, it is marked dead and i reenters the request phase.

Receive Phase

1. If a request_parent message arrives from m then:
   (a) If i has a free child slot, it accepts m's request and sends back an ack message
   (b) If i has exhausted its allotted number of child slots then:
      1) i scans its children for a late child as defined below. If one is found then
         (a) the child is marked as dead and removed
         (b) this makes a child slot available
         (c) i sends m an ack message
      If a late child is not found then
      2) i finds the child k with highest rank.
      3) If rank(k) > rank(m) then:
         a) i accepts m as a child in place of k and sends m an ack message.
         b) i sends k a disengage message containing rank(m) as a parameter.
      4) If rank(k) < rank(m) then:
         a) i locates in its list of computers the computer n with the lowest rank satisfying rank(n) > rank(i).
         b) i sends m a noack message containing rank(n) as a parameter.
2. If a disengage message with parameter r arrives then:
   a) i clears its parent field
   b) i removes the computer with ranks outside the range [rank(i) − 1,r] from CRi and starts looking for a new parent, ie enters request phase.
3. If ack arrives from candidate parent the computer records the candidate parent as a true parent.
4. If a noack message with parameter r arrives then:
   (a) i clears its candidate parent field
   (b) the computer executes step 2(b) above.

Figure 1a is a flow diagram showing the request phase of the tree generation and Figures 1b, 1c and 1d are flow diagrams showing the receive phase of the tree generation.

As an example of tree generation, consider 5 computers with ranks 0 to 4 where each can accept two children. Due to random choice of parents the method can reach the stage shown in Figure 2a.

Computer 1 can choose only computer 0 as a parent but computer 0 has already two children. Therefore, the method replaces child 3 of computer 0 with computer 1 and sends computer 3 a disengage message with parameter 1. Thus the situation pictured in Figure 2b is reached.

Node 3 searches for a parent in the range 2 to 1 and picks, say 1 as parent. The resulting tree appears in Figure 2c.

If all nodes are alive and there are no late children then the tree generation terminates with one tree.

The above tree generation may build unbalanced trees or trees with many nodes having only one child. However, it has been found that the load balancing method is insensitive to the topology of the tree.

Tree maintenance is required to detect dead nodes, to add new nodes, or to reconnect rebooted nodes. It is necessary because nodes fail and nodes are sometimes shut down for maintenance. In this embodiment of the invention such cases are handled in the following way.

1. Each computer periodically sends an update_up message to its parent and waits for an update_down message.
2. If no response arrives within a specified period of time the parent is marked dead and the computer looks for a new parent.
3. If an update_up message arrives from a child the node responds with an update_down message.
4. If no update_up message arrives from a child within a specified period of time the child is marked late.
5. If an update_up message arrives from a late child then the late flag for that child is reset and the node responds with an update_down message.
6. When a computer is rebooted or added to the network it looks for a parent, ie it enters the request phase.
7. Each computer flags computers already tried and those marked dead. Periodically these flags are cleared, and CR is reset to contain all computers with lesser rank. Thus rebooted nodes can be probed again.

As an example, consider a tree of 18 nodes labelled a to r as shown in Figure 3a.

If node h fails, then after a predefined period of time, its parent c will mark it as late and ignore it since it did not receive an update_up message. Nodes k and r will start looking for new parents since no update_down message arrived. The network now comprises a number of separate trees as shown in Figure 3b.

k and r look for a new parent in their FP. If their FP is empty they choose a parent at random. The new tree may look like the one in Figure 3c, with node k having node g as its new parent and node r having node c as its new parent.

The tree generation together with the tree maintenance mechanism ensure that the computers in the network generally will be arranged in a single tree structure. To see that this is true in the case of a node failing consider an arbitrary tree in which node k fails or is otherwise inoperational. k's parent will mark it as late and k's children will mark it as dead. Assume m was a child of k. Therefore, m looks for a new parent. There are three possibilities:

1. m finds a node which can accept it as a child. Thus, a single tree is formed.
2. m picks k's parent as a candidate and there are no free slots. In this case the slot of k marked as late is used and k is marked as dead. Again, a single tree is formed.
3. m does not find a node which can accept it as child.
This can happen only if the nodes which would have accepted it (there is at least one, ie m−l) are marked as already tried or as dead. In this case the network temporarily comprises a number of sub-trees rather than a single tree. This state will end when all these flags are reset (step 7 of the maintenance mechanism) and then m will try again and will find a parent.
The above reasoning is applicable for all the children of k.

EXCHANGE AND MAINTENANCE OF INFORMATION

The number of jobs on the run queue is generally accepted as a suitable load metric. A node is considered overloaded if the length of the run queue exceeds a predefined threshold, overload_mark. Since the quantity that is really of interest is how far the node is from its overload mark, a slightly different metric is used in this embodiment of the invention. In this embodiment of the invention load is defined as length_of_run−queue − overload_mark. For example, if $overload\_mark_A = 10$ and $overload\_mark_B = 8$ and the length of run queue of nodes A and B are 5 and 4 respectively, then the $load_A = -5$ and $load_B = -4$. Thus, the load of B is higher than that of A.

In the following this metric will be used for load and whenever the term load is used, length_of_run_queue − overload_mark is meant.

Each computer in the network stores a sorted vector which holds information on other computers in the network. Each entry in the vector contains:

1) the rank of the computer
2) the local load of the computer
3) the distance in terms of the number of edges or links the load information has traversed

4) the last node that propagated this information (last node field)

The length of the load vector, ie the number of entries it contains, may vary from node to node.

Load information is distributed over the network in the following way.

1. Periodically each computer samples its load L, sorts its load vector using the new value of L, and sends the load vector to its parent in an update_up message. The update_up message is also used as the indication that the node is operational for the purposes of tree maintenance.
2. When an update_up message is received by node m from node k,
   a The distance of each entry in the received load vector is incremented by 1.
   b All entries in the received load vector which originated from receiving node m are deleted.
   c All entries in the load vector which originated from the sending node k are deleted.
   d The distances of entries (computers) which appear in both the local vector and the received vector are compared:
      1. If the distance of local entry is greater or equal to the distance of the received entry then the local entry is deleted.
      2. If the distance of the received entry is greater than the distance of the local entry then the received entry is deleted.
   e The last node field in the entries of the received vector is set to be the sending node k.
   f The received load vector is merged with the local load vector.
3. A parent sends its new load vector to the child who sent the update_up message. This message is an update_down message which again is used as an indication that the parent is operational in the tree maintenance message.
4. When an update_down message arrives, the receiving child processes the received load vector in the same way as described in step 2.

The update_down messages propagate information received by each node from the nodes to which they are linked in the tree structure. Thus children share information by passing it up to the parent who passes it down to the other children. By propagating load information up and down the tree, information on under-loaded nodes in another sub-tree can reach any node. Due to the fixed size of the load vector, each computer holds information on a sub-set of the computers that are least loaded.

The load information in a load vector is only an approximation of the real load on any particular computer, since by the time it reaches another node the load of the originating computer may have changed.

The 'update interval' is defined as the time interval between successive load distributions; ie update_up messages. This interval significantly influences the results of load balancing. Too large a value renders the load information of the current node in other nodes inaccurate. Migration decisions based on this information result in many rejections due to the inaccuracy of the information. Too small a value; ie a higher update rate, increases the overhead of the method and response time is degraded. The update rate (1/update_interval) should be proportional to the rate of generation of local processes and should change as this rate changes. In addition there is another factor which has to be considered. It has to be ensured that the update rate of the current node does not slow down the propagation of information from its children to its parent and vice versa. This wills happen if the rate of the current node is low and that of its children or its parent is high. Therefore, the update rates of the immediate neighbours have also to be considered.

The effective_rate of a node is defined as the result of a rate calculation function based on these two factors.

In this embodiment of the invention the following function is used. Initially the effective rate of each node is set to the birth rate of local processes. For each node p the following quantities are computed:

a) $rate_1$ = local_birth_$rate_p$

Local birth $rate_p$ is computed as an average over a sliding time window.

b) $rate_2 = \max\{effective\_rate_n\}/\alpha$

for all immediate neighbours n of p. α is an attenuation factor determined by computer simulation experiments as described below.

The effective_rate of p is given by:

$$effective\_rate_p = \max\{rate_1, rate_2\}$$

In computer simulation experiments it was found that values of α in the range 1.4 to 2.0 gave good results for networks with evenly distributed load and networks with regions of high load.

When a computer is overloaded it searches its local load vector for the first entry indicating a computer that can accept extra load, and transfers one or more tasks directly to it. The load entry of the target

computer in the load vector is updated and the load vector is re-sorted. If the target computer cannot for whatever reasons accept extra load, the sender is notified. In this case the sender deletes that entry from the load vector, retrieves the next entry and use it as a new target. If no node can accept extra load then no migration is initiated.

Thus, the location policy; ie, finding a target node, depends on the sort criterion used to sort the load vector. Since load information is propagated in hops over the tree, the farther away a node is, the less accurate is its information. Therefore, in this embodiment of the evaluation the load vector is sorted according to load and distance. Entries are first sorted according to their load. Entries with equal load are then sorted in ascending order of distance.

A node which is lightly loaded for a long time may appear as the first entry in the load vector of many nodes. This may cause many nodes to migrate processes to that node, thus flooding it and creating a bottleneck. Two approaches can be used to avoid this problem.

An equivalence set of nodes can be defined in the load vector as those nodes with the same load and distance. For example all nodes with load −1 and distance 2 are in the the same equivalence set. In this case, the sort criterion ensures that all the nodes in the same equivalence set are grouped together in the vector. Before a load vector is distributed (by update_up or update_down message) the first equivalence set in the vector undergoes permutation. This reduces the probability that the same node wills appear in many load vectors.

In addition, whenever a lightly-loaded node appears first in its own load vector for the first time, a counter is attached to the entry and initialised to be equal to the load of the lightly loaded node which is always less than zero. Whenever the vector is distributed, if that entry is still first, then the counter is incremented. If the counter becomes positive that entry is removed from the vector. Thus the number of nodes which are aware of the lightly loaded node is limited and the probability of flooding reduced.

Initiation of process migration depends on the frequency the local load is checked and on the value of the local load. The decision when to check a node's load status depends on the chosen policy. There are a number of possible policies, including the following

1. Periodic — the local load is tested after sending update_up message. ie, the local load sampling is synchronised with the update_up message.
2. Event driven — The local load is tested whenever a local process is added to the run queue.

The decision when to declare a computer overloaded, ie where to place the overload mark, influences the performance of the method. Placing the overload mark too low may cause too many migrations without improving response time. Placing it too high may degrade response time. The overload mark value needs to be optimised for any particular system and the value may be different for computers in the network having different load capacities or for different job type profiles. The optimum value or values can be established in any given situation by the use, for example, of appropriate computer simulation experiments or by a process of trial and error.

A controlled clustering of nodes can be achieved by an appropriate assignment of rank in the configuration files of each node.

For example, if there are eleven computers a to k and it is required to form clusters of four and seven computers as follows,

a b c d	e f g h i j k
1st cluster	2nd cluster

the following steps are performed to rank the nodes.

1. Each cluster is ranked separately. For example

a b c d	e f g h i j k
1 2 0 3	5 6 0 1 2 3 4

2. Each cluster is taken in turn, and rank of each node is increased by the number of nodes in previous clusters. For example if the 2nd cluster is chosen first.

a	b	c	d		e	f	g	h	i	j	k
8	9	7	10		5	6	0	1	2	3	4

Cluster configuration is achieved by assigning different ranking files to the computers. Each computer except the one with lowest rank in a cluster, stores a ranking list that contains ranks of nodes in its cluster and not those outside the cluster. The computer with the lowest rank stores the ranking list with nodes in other clusters.

Nodes in a cluster will therefore connect to each other since they are not aware of other nodes. Only the node with the lowest tank will connect to a node in another cluster, because it cannot connect to another node in the cluster. Figure 4a shows a sample tree formed from the nodes in the above example.

If the maximum number of children for c is three, and the favoured parent lists of a, b and d contain only c, the tree described in Figure 4b may result.

Placing j in the favoured parent list of c may generate the tree described in Figure 4c.

Thus clustering is achieved in the embodiment of the invention by judicious choice of ranking files and favoured parents lists for each node.

Whilst the invention has been described in terms of an embodiment using a sender-initiated load balancing policy in which an overloaded node initiates the transfer of tasks, it will be clear to those skilled in the art that the invention could equally well be applied to a receiver-initiated scheme in which a lightly loaded computer initiates transfer of tasks from a heavily loaded computer to itself.

There has been described a method of operating a computer in a network and operating a network of computers using an improved load balancing technique which is scalable, fault tolerant, flexible and supports clustering thus making it suitable for use in networks having very large numbers of computers.

Claims

1. A method of operating a computer in a network of computers, the method comprising:
generating logical links between the computer and other computers in the network so that a tree structure is formed, the computer being logically linked to one computer higher up the tree and a number of computers lower down the tree; and
maintaining in the computer stored information regarding the current load on the computer and the load on at least some of the other computers in the network by causing the computer periodically to distribute the information to the computers to which it is logically linked, to receive from said computers similar such information and to update its own information in accordance therewith, so that the information can be used to determine computers in the network that can accept extra load.

2. A method as claimed in claim 1 comprising the step of, when the computer is overloaded, using the information to determine a computer that can accept extra load and transferring at least one task to that computer.

3. A method as claimed in claim 1 or claim 2 comprising, if the computer higher up the tree, to which the computer is logically linked fails or is otherwise inoperative, generating a new logical link to another computer in the network which has capacity for accepting new downward links.

4. A method as claimed in any preceding claim wherein the periodic distribution of load information is used by the computer to determine whether or not the computers to which it is linked in the tree, are operational, the computer being marked, if one of the computers lower in the tree to which it is linked is not operational, as having capacity to accept new downward links.

5. A method as claimed in any preceding claim wherein the information stored in the computer contains a number of entries, each entry containing information regarding
   (i) the load on one of the computers in the network,
   (ii) the number of links in the tree separating that other computer from the computer, and
   (iii) the one of the computers, which are linked to the computer, from which the entry was last received:
and wherein when the computer receives the similar information from a computer to which it is linked the following steps are performed:

(a) the number of links value in each entry of the received information is incremented by one;
(b) entries in the received information which originated from the computer are deleted;
(c) entries in the information already stored in the computer which were received from the sending computer are deleted;
(d) the received information is merged with the information already stored in the computer;
(e) the merged information is sorted in ascending order of load, entries with equal load being sorted in ascending order of number of links separation from the computer.

6. A method as claimed in claim 5 wherein before each distribution of information entries in the information relating to computers which have the same load and number of links separation from the computer are randomly permuted.

7. A method as claimed in claim 5 or claim 6 wherein if an entry corresponding to the computer appears first in its own sorted information a counter is attached to the entry and initialised to negative value of the spare load capacity of the computer, the counter being incremented, if that entry is still first, whenever the information is distributed, and if the counter becomes positive that entry is removed from the information, whereby the probability that the same entry will appear first in the information stored in different computers is reduced.

8. A method as claimed in any preceding claim wherein the period of the periodic distribution of the information to a computer higher up the tree depends on the rate at which new tasks are created in the computer

9. A method as claimed in claim 8 wherein the period of the periodic distribution depends on the rate at which new tasks are created in the computers to which the computer is logically linked in the tree structure.

10. A method as claimed in any preceding claim wherein the distribution of the information to a computer lower in the tree takes place in response to receipt by the computer of the similar information from the computer lower in the tree.

11. A method of operating a network of computers, comprising operating each computer using a method as claimed in any preceding claim.

12. A method as claimed in claim 11 wherein the generation of the logical links is achieved by assigning a rank to each computer, no two computers being assigned the same rank, each computer being linked to one computer of lower rank and a number of computers of higher rank to form the tree structure.

13. A computer capable of operating in a network of similar such computers, the computer comprising:
means identifying computers in the network to which the computer is logically linked in a tree structure;
means for storing information regarding the load on the computer and at least some of the other computers in the network;
means for sending said information to the computers to which the computer is logically linked;
means for receiving similar information from said computers to which the computer is logically linked and updating the stored information in accordance therewith;
means for selecting one of the other computers in the network having spare load capacity from said information and transferring tasks to the selected computer.

14. A computer as claimed in claim 13 including:
means for accessing a stored list of all the computers in the network;
means for selecting a computer from this list as a candidate neighbouring computer for linking in the tree structure;
means for sending a message to the selected computer indicating a link request;
means for establishing a logical link with the selected computer by updating the identifying means if a positive response from the selected computer is received;
means from receiving messages from other computers indicating that a link is requested;
means for determining, on receipt of such a link request message, whether or not there is capacity for establishing a downward link and sending a positive response to the sender of the link request

message if there is such capacity and updating the identifying means accordingly.

15. A network of computers comprising a plurality of computers as claimed in claim 13 or claim 14.

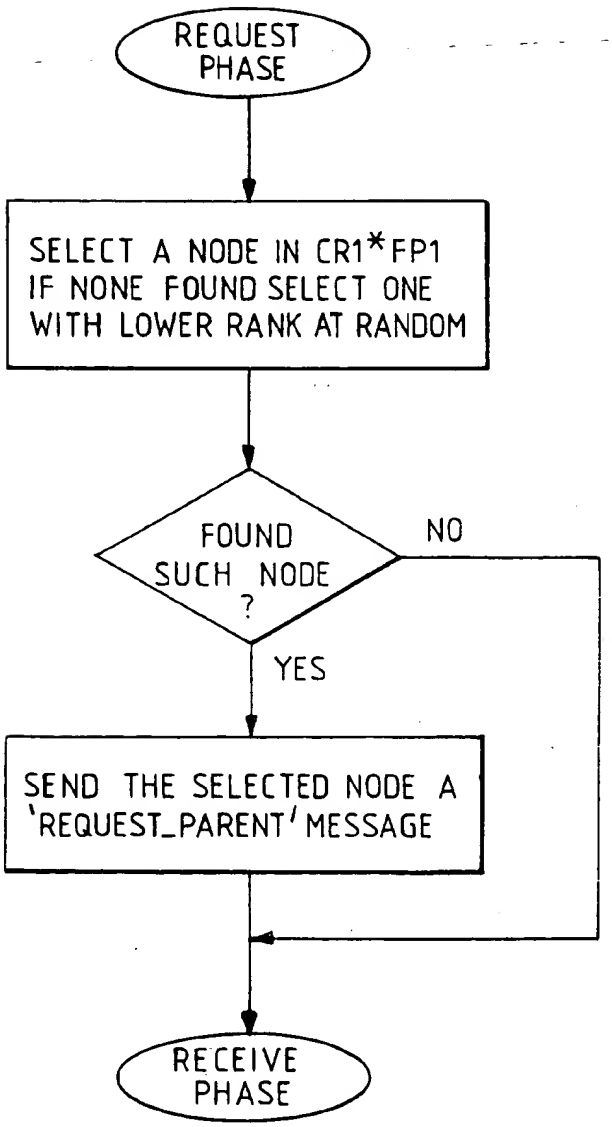
REQUEST PHASE
SELECT A NODE IN CR1*FP1 IF NONE FOUND SELECT ONE WITH LOWER RANK AT RANDOM
FOUND SUCH NODE ?
NO
YES
SEND THE SELECTED NODE A 'REQUEST_PARENT' MESSAGE
RECEIVE PHASE

FIG.1a

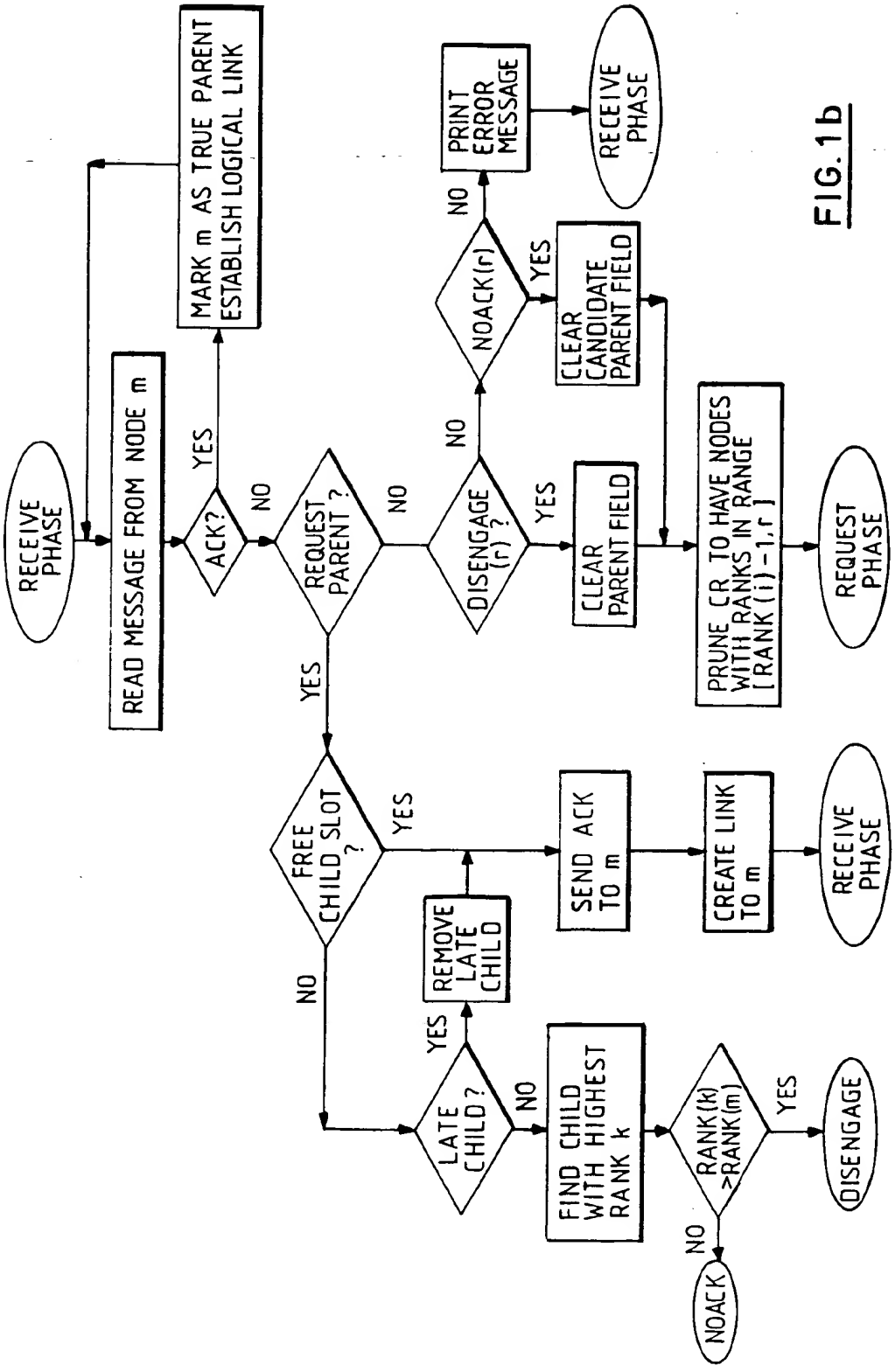
RECEIVE PHASE
READ MESSAGE FROM NODE m
ACK?
YES
MARK m AS TRUE PARENT ESTABLISH LOGICAL LINK
NO
REQUEST PARENT ?
NO
DISENGAGE (r) ?
NO
NOACK(r)
NO
PRINT ERROR MESSAGE
RECEIVE PHASE
YES
CLEAR CANDIDATE PARENT FIELD
YES
CLEAR PARENT FIELD
PRUNE CR TO HAVE NODES WITH RANKS IN RANGE [RANK (i) -1, r]
REQUEST PHASE
YES
FREE CHILD SLOT ?
YES
SEND ACK TO m
CREATE LINK TO m
RECEIVE PHASE
NO
LATE CHILD ?
YES
REMOVE LATE CHILD
NO
FIND CHILD WITH HIGHEST RANK k
RANK(k) >RANK(m)
YES
DISENGAGE
NO
NOACK

FIG. 1b

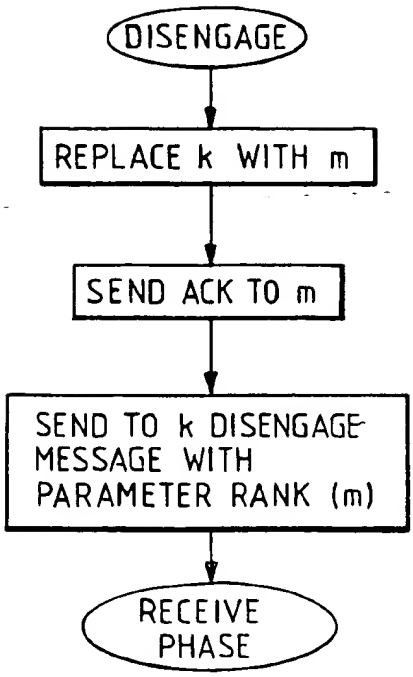
DISENGAGE
REPLACE k WITH m
SEND ACK TO m
SEND TO k DISENGAGE MESSAGE WITH PARAMETER RANK (m)
RECEIVE PHASE

FIG. 1c

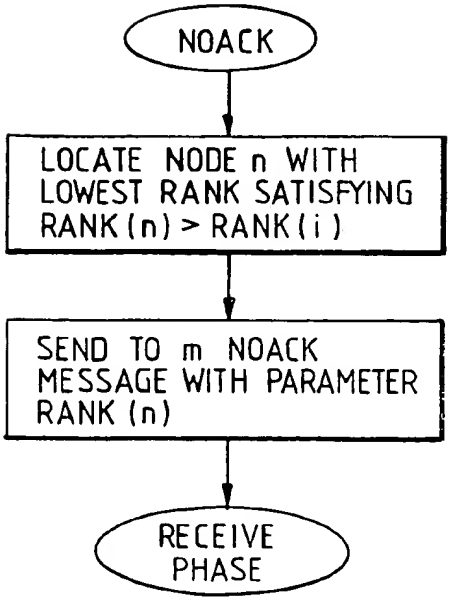
NOACK
LOCATE NODE n WITH LOWEST RANK SATISFYING RANK(n) > RANK(i)
SEND TO m NOACK MESSAGE WITH PARAMETER RANK (n)
RECEIVE PHASE

FIG. 1d

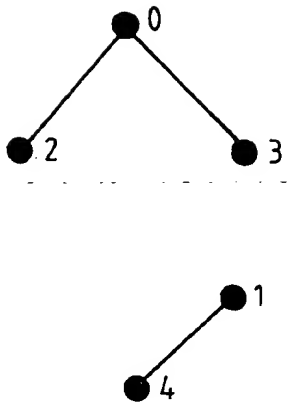

FIG. 2a

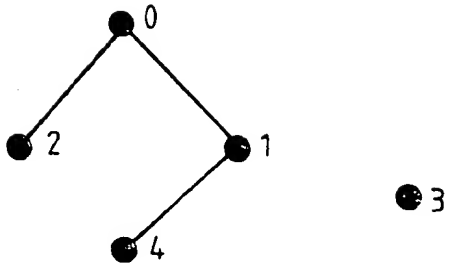

FIG. 2b

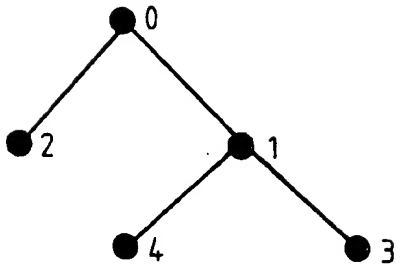

FIG. 2c

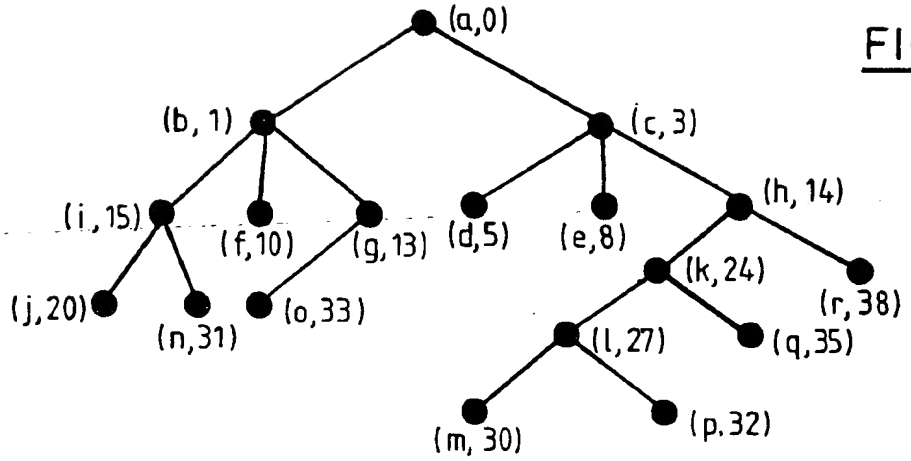

FIG. 3a

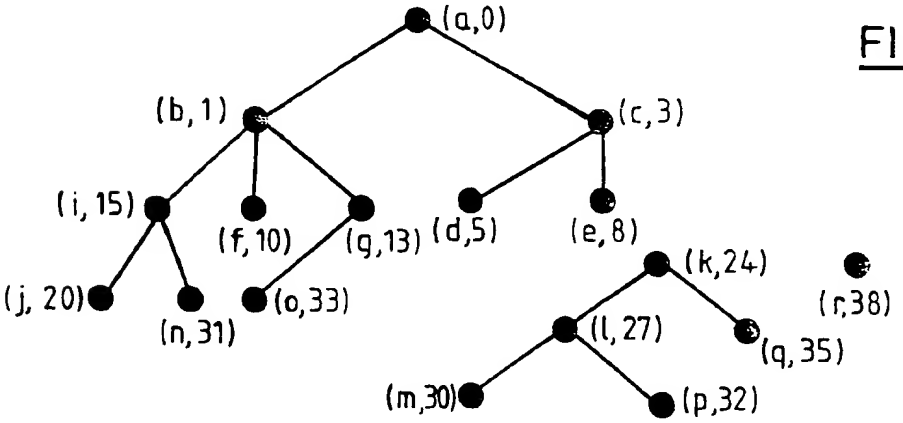

FIG. 3b

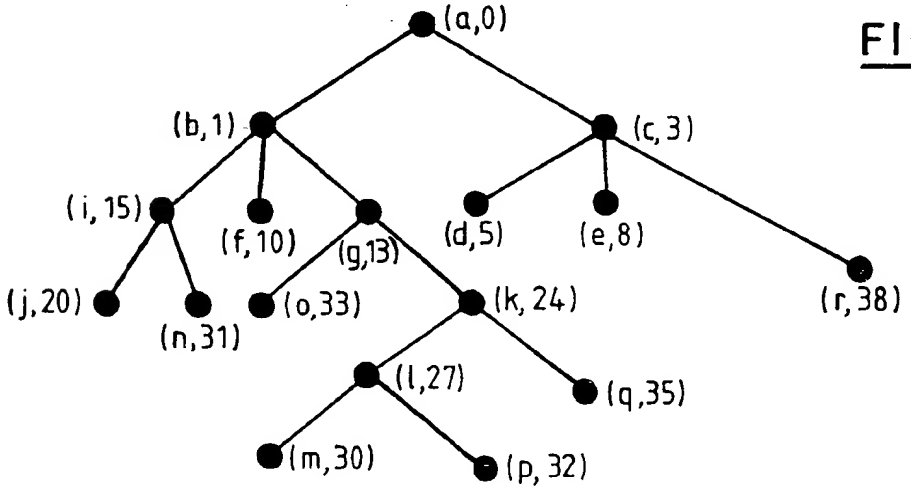

FIG. 3c

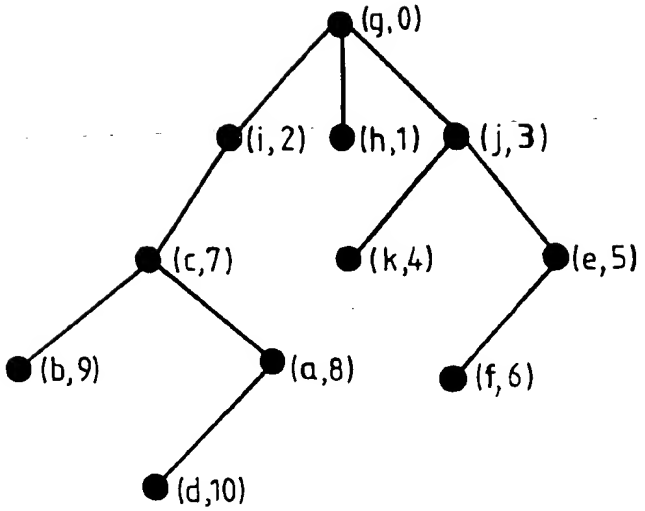

FIG. 4a

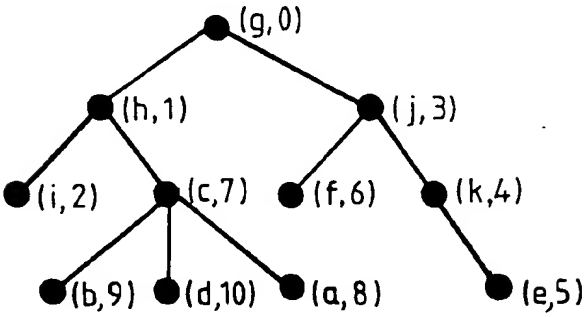

FIG. 4b

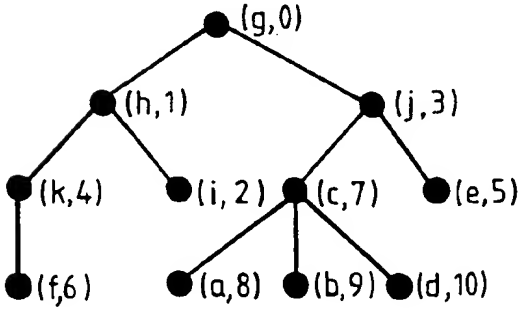

FIG. 4c

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 540 151 A3**

(12) **EUROPEAN PATENT APPLICATION**

(21) Application number: 92308137.6

(51) Int. Cl.5: G06F 9/46

(22) Date of filing: 08.09.92

(30) Priority: 31.10.91 IL 99923

(43) Date of publication of application:
05.05.93 Bulletin 93/18

(84) Designated Contracting States:
DE FR GB

(88) Date of deferred publication of the search report:
13.10.93 Bulletin 93/41

(71) Applicant: International Business Machines Corporation
Old Orchard Road
Armonk, N.Y. 10504(US)

(72) Inventor: Allon, David
49/8 Meir Nakar Street
Jerusalem(IL)
Inventor: Bach, Moshe
Trumpeldor Street 5a
Haifa(IL)
Inventor: Moattl, Yosef
68/56 Hanita Street
Haifa(IL)
Inventor: Teperman, Abraham
46 Havlva Reich Street
Haifa(IL)

(74) Representative: Burt, Roger James, Dr.
IBM United Kingdom Limited
Intellectual Property Department
Hursley Park
Winchester Hampshire SO21 2JN (GB)

(54) **Method of operating a computer in a network.**

(57) A method is described of operating a computer in a network of computers using an improved load balancing technique, the method comprising: generating logical links between the computer and other computers in the network so that a tree structure is formed, the computer being logically linked to one computer higher up the tree and a number of computers lower down the tree; maintaining in the computer stored information regarding the current load on the computer and the load on at least some of the other computers in the network by causing the computer periodically to distribute the information to the computers to which it is logically linked, to receive from said computers similar such information and to update its own information in accordance therewith, so that the information can be used to determine a computer in the network that can accept extra load. A sender-initiated embodiment of the invention includes the further step of, when the computer is overloaded, using the information to determine a computer that can accept extra load and transferring at least one task to that computer. The load balancing technique is scalable, fault tolerant, flexible and supports clustering thus making it suitable for use in networks having very large numbers of computers.

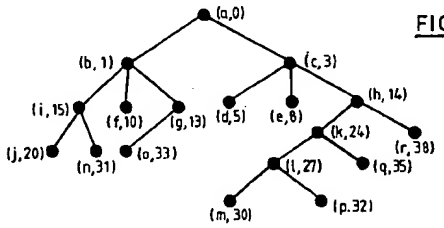

FIG. 3a

EP 0 540 151 A3

European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 92 30 8137

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int. Cl.5)
A	IEEE TRANSACTIONS ON AEROSPACE AND ELECTRONIC SYSTEMS vol. 26, no. 3, May 1990, NEW YORK US pages 511 - 516 Y.C.CHENG ET AL. 'Distributed Computation for a Tree Network with Communication Delays' * figure 1 * * page 511, right column, line 1 - line 27 * ---	1,13	G06F9/46
A	SYSTEMS & COMPUTERS IN JAPAN vol. 19, no. 6, June 1988, SILVER SPRING, MD, US pages 35 - 48 M.TAKESUE 'A Distributed Load-Balancing System and its Application to List-Processing Oriented Data-Flow Machine DFM' * page 35, left column, line 1 - page 36, left column, line 54 * * figure 1 * ---	1,13	TECHNICAL FIELDS SEARCHED (Int. Cl.5)
P,A	IEEE TRANSACTIONS ON COMPUTERS vol. 41, no. 3, March 1992, NEW YORK US pages 257 - 273 N.S.BOWEN ET AL. 'On the Assignment Problem of Arbitrary Process Systems to Heterogenous Distributed Computer Systems' * abstract * * page 265, right column, line 1 - page 267, right column, line 10 * * figures 1,9,12 * -----	1,13	G06F

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	19 AUGUST 1993	SCHARFENBERGER B.

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P0001)

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 540 151 B1

(12) **EUROPEAN PATENT SPECIFICATION**

(45) Date of publication and mention of the grant of the patent:
25.11.1998 Bulletin 1998/48

(51) Int Cl.[6]: **G06F 9/46**

(21) Application number: **92308137.6**

(22) Date of filing: **08.09.1992**

(54) Method of operating a computer in a network

Verfahren zum Betrieb eines Rechners in einem Netz

Méthode d'opération d'un ordinateur dans un réseau

(84) Designated Contracting States:
DE FR GB

(30) Priority: **31.10.1991 IL 99923**

(43) Date of publication of application:
05.05.1993 Bulletin 1993/18

(73) Proprietor: **International Business Machines Corporation**
Armonk, N.Y. 10504 (US)

(72) Inventors:
- **Allon, David**
  Jerusalem (IL)
- **Bach, Moshe**
  Haifa (IL)
- **Moatti, Yosef**
  Haifa (IL)
- **Teperman, Abraham**
  Haifa (IL)

(74) Representative: **Burt, Roger James, Dr.**
IBM United Kingdom Limited
Intellectual Property Department
Hursley Park
Winchester Hampshire SO21 2JN (GB)

(56) References cited:
- **IEEE TRANSACTIONS ON AEROSPACE AND ELECTRONIC SYSTEMS vol. 26, no. 3, May 1990, NEW YORK US pages 511 - 516 Y.C.CHENG ET AL. 'Distributed Computation for a Tree Network with Communication Delays'**
- **SYSTEMS & COMPUTERS IN JAPAN vol. 19, no. 6, June 1988, SILVER SPRING, MD, US pages 35 - 48 M.TAKESUE 'A Distributed Load-Balancing System and its Application to List-Processing Oriented Data-Flow Machine DFM'**
- **IEEE TRANSACTIONS ON COMPUTERS vol. 41, no. 3, March 1992, NEW YORK US pages 257 - 273 N.S.BOWEN ET AL. 'On the Assignment Problem of Arbitrary Process Systems to Heterogenous Distributed Computer Systems'**

Note: Within nine months from the publication of the mention of the grant of the European patent, any person may give notice to the European Patent Office of opposition to the European patent granted. Notice of opposition shall be filed in a written reasoned statement. It shall not be deemed to have been filed until the opposition fee has been paid. (Art. 99(1) European Patent Convention).

EP 0 540 151 B1

Printed by Jouve, 75001 PARIS (FR)

Description

The invention relates to methods of operating computers in networks and to computers capable of operating in networks.

Local area networks of computers are an indispensable resource in many organisations. One problem in a large network of computers is that it is sometimes difficult to make efficient use of all the computers in the network. Load sharing or balancing techniques increase system throughput by attempting to keep all computers busy. This is done by off-loading processes from overloaded computers to idle ones thereby equalising load on all machines and minimising overall response time.

Load balancing methods can be classified according to the method used to achieve balancing. They can be 'static' or 'dynamic' and central' or 'distributed'.

In static load balancing, a fixed policy - deterministic or probabilistic - is followed independent of the current system state. Static load balancing is simple to implement and easy to analyse with queuing models. However, its potential benefit is limited since it does not take into account changes in the global system state. For example, a computer may migrate tasks to a computer which is already overloaded.

In dynamic schemes the system notes changes in its global status and decides whether to migrate tasks based on the current state of the computers. Dynamic policies are inherently more complicated than static policies since they require each computer to have knowledge of the state of other computers in the system. This information must be continuously updated.

In a central scheme one computer contains the global state information and makes all the decisions. In a distributed scheme no one computer contains global information. Each computer makes migration decisions based on the state information it has.

Load-balancing methods can be further classified as 'sender-initiated' and 'receiver-initiated'. In sender-initiated policies an overloaded computer seeks a lightly loaded computer. In receiver-initiated policies lightly loaded computers advertise their capability to receive tasks. It has been shown that if costs of task-transfer are comparable under both policies then sender-initiated strategies outperform receiver-initiated strategies for light to moderate system loads, while receiver-initiated schemes are preferable at high system loads.

Conventionally, a dynamic load balancing mechanism is composed of the following three phases:

1. Measuring the load of the local machine.

2. Exchanging local load information with other machines in the network.

3. Transferring a process to a selected machine.

Local load can be computed as a function of the number of jobs on the run queue, memory usage, paging rate, file use and I/O rate, or other resource usage. The length of the run queue is a generally accepted load metric.

The receipt of load information from other nodes gives a snapshot of the system load. Having this information, each computer executes a transfer policy to decide whether to run a task locally or to transfer it to another node. If the decision is to transfer a task then a location policy is executed to determine the node the task should be transferred to.

If a load balancing method is to be effective in networks having large numbers of computers it must be scalable, in the sense that network traffic increases linearly with the size of the network, flexible so that computers can be easily added to or removed from the network, robust in the event of failure of one or more of the computers and must support to some degree clustering of the computers.

Centralised methods, though attractive due to their simplicity are not fault-tolerant. The central computer can detect dead nodes and it takes one or two messages to re-establish connection. However, if the central node fails, the whole scheme falls apart. Load balancing configuration can be re-established by maintaining a prioritised list of alternative administrators in each node or by implementing an election method to elect a new centralised node. When a central node fails, other nodes switch to the new central node. This enhancement, however, increases the management complexity and cost of the method.

In addition, management overhead for a centralised method is unacceptably large. As the number of nodes increases, the time spent by the central node in handling load information increases, and there must come a point at which it will overload.

Several prior art load balancing methods are both dynamic and distributed.

For example, in the method described in Barak A. and Shiloh A. SOFTWARE - PRACTICE AND EXPERIENCE, 15(9):901-913, September 1985 [R1] each computer maintains a fixed size load vector that is periodically distributed between computers. Each computer executes the following method. First, the computer updates its own load value. Then, it chooses a computer at random and sends the first half of its load vector to the chosen computer. When receiving

a load vector, each computer merges the information with its local load vector according to a predefined rule.

A problem with this method is that the size of the load vector has to be chosen carefully. Finding the optimal value for the load vector is difficult and it must be tuned to a given system. A large vector contains load information for many nodes. Thus, many nodes will know of a lightly loaded node, increasing the chance that it will quickly receive many migrating processes and will overload. On the other hand, the size of the load vector should not be so small that load values do not propagate through the network in a reasonable time. For large number n of nodes in the network, the expected time to propagate load information through the network is O(log n).

The method described in R1 can handle any number of computers after tuning the length of the load vector. Therefore, the method is scalable. However, it is flexible only up to a point. Adding a small number of nodes to the network does not require any change. Adding a large number of nodes requires changing the size of the load vector in all computers, thus increasing the administrative overhead. The method is fault-tolerant and continues to work in spite of single failures, however there is no built-in mechanism for detecting dead nodes and updating information on them in the load vectors. Thus information about failed nodes is not propagated and other nodes may continue to migrate processes to them, with the result of decreased response time.

The number of communications per unit time is O(n). However, the method does not support clustering. When a node wants to off-load processes to another node it chooses a candidate from its load vector. Since the information on nodes in the load vector of node p is updated from random nodes, the set of candidates for off-loading is a random subset of n and not a controlled set for p.

In Lin F. C. H. and Keller R. M. PROCEEDINGS OF THE 6TH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 329-336, May 1986 [R2], a distributed and synchronous load balancing method using a gradient model is disclosed. Computers are logically arranged in a grid and the load on the computers is represented as a surface. Each computer interacts solely with its immediate neighbours on the grid. A computer with high load is a 'hill' and an idle computer is a 'valley'. A neutral computer is one which is neither overloaded nor idle. Load balancing is a form of relaxation of the surface. Tasks migrate from hills towards valleys and the surface is flattened. Each computer computes its distance to the nearest idle computer and uses that distance for task migration.

An overloaded node transfers a task to its immediate neighbour in the direction of an idle node. A task continues to move until it reaches an idle node. If a former idle node is overloaded when a task arrives, the task moves to another idle node.

This method, though scalable, is only partially flexible. It is easy to add or remove nodes at the edge of the grid, but it is difficult to do so in the middle since the grid is fixed. To do this, reconfiguration is required, which increases the overhead inherent in the method. Detection of dead nodes is not easy. Since only changes of the state of a node are sent to its neighbours, a node's failure remains undetected until a job is transferred to it. Late detection delays migrating processes and, therefore, increases overall response time. In this method clustering is not supported. Each node transfers processes to one of its immediate neighbours which may transfer it in a different direction. The overloaded node has no control as to where the off-loaded processes will eventually execute. Management overhead to start the grid is low and the number of messages is O(n). Administrative overhead for reconfiguration is high since it requires changes in all neighbouring nodes for a failed node. In this method node overhead is high and network traffic increases because tasks move in hops rather than directly to an idle node and it takes long time to migrate a process.

The buddy set algorithm proposed in Shin K. G. and Chang Y. C. IEEE TRANSACTIONS ON COMPUTERS, 38 (8), August 1989 [R3], aims for very fast load sharing. For each node two lists are defined: Its buddy set, the set of its neighbours, and its preferred list, an ordered list of nodes in its buddy set to which tasks are transferred. Each node can be in one of the following states: under-loaded, medium-loaded and overloaded. Each node contains the status of all nodes in its buddy set. Whenever the status of a node changes, it broadcasts its new state to all nodes in its buddy set. The internal order of preferred lists varies per node in order to minimise the probability that two nodes in a buddy set will transfer a task to the same node. when a node is overloaded it scans its preferred list for an underloaded node and transfers a task to that node. Overloaded nodes drop out of preferred lists. Buddy sets and their preferred lists overlap, so processes migrate around the network.

Again, this method, though scalable in that the number of messages increases linearly with the number of computers in the network n, is not flexible. Adding or removing nodes from the network requires recomputation of the preferred lists in all nodes of the related buddy set. If a node is added to more than one buddy set the recomputation must be done for each node in each buddy set. Detection of dead nodes is difficult for the same reason as in the gradient model described in R2. Clustering is supported but reconfiguration is expensive since it requires recomputation of new buddy sets and preferred lists, as for adding and removing nodes. The administrative overhead inherent in the method is therefore high.

Cheng Y.-C. and Robertazzi T. G. in IEEE TRANSACTIONS ON AEROSPACE AND ELECTRONIC SYSTEMS, vol. 26, no. 3, 511-516, May 1990 [R4], disclose a method of operating a computer in a network of computers, the method comprising the step of generating logical links between the computer and other computers in the network so that a tree structure is formed. The logical links are such that the computer is logically linked to one computer higher

up the tree and a number of computers lower down the tree.

According to a first aspect of the present invention there is provided a method of operating a computer in a network of computers characterised by:

maintaining in the computer stored information regarding the current load on the computer and the load on at least some of the other computers in the network by causing the computer periodically to distribute the information to the computers to which it is logically linked, to receive from said computers similar such information and to update its own information in accordance therewith, so that the information can be used to determine computers in the network that can accept extra load; wherein the information stored in the computer contains a number of entries, each entry containing information regarding:

(i) the load on one of the computers in the network,

(ii) the number of links in the tree separating that other computer from the computer, and

(iii) the one of the computers, which are linked to the computer, from which the entry was last received;

and wherein when the computer receives the similar information from a computer to which it is linked the following steps are performed:

(a) the number of links value in each entry of the received information is incremented by one;

(b) entries in the received information which originated from the computer are deleted;

(c) entries in the information already stored in the computer which were received from the other computer are deleted;

(d) the received information is merged with the information already stored in the computer;

(e) the merged information is sorted in ascending order of load, entries with equal load being sorted in ascending order of number of links separation from the computer.

The invention is applicable to both sender-initiated and receiver-initiated load balancing. An embodiment of the invention using a sender-initiated load balancing technique includes the further step of, when the computer is overloaded, using the information to determine a computer that can accept extra load and transferring at least one task to that computer.

There is further provided a method of operating a network of computers by operating each computer using the above method.

The generation of the logical links can be achieved by assigning a rank to each computer, no two computers being assigned the same rank, each computer being logically linked to one computer of lower rank and a number of computers of higher rank to form the tree structure.

The tree structure can be maintained if a computer fails, or is otherwise inoperative, by generating new logical links between each of the computers lower down the tree to which the failed computer was linked and other computers, which have capacity for accepting new downward links.

An improved load balancing technique is employed which has the property of scalability by limiting the communication load on each computer and the knowledge of the network topology which is required to be stored in each computer. The actions taken by each computer are thus independent of the total number of computers in the network.

Since the tree structure is dynamically built and maintained, the method is also flexible. Changing a tree configuration requires simply changing the ranking of the reconfigured computers. The new ranking files have to be consistent with existing ranking files to avoid cycles. A computer is reconfigured by logically disconnecting it so that it appears inoperative to its neighbours, replacing its configuration file, and letting it reconnect to the tree. A new configuration is achieved without disturbing other nodes on the tree and without disrupting local services.

Detection and reconnection of failed nodes is also provided for. If a computer higher up the tree does not respond within a certain time each higher ranked computer previously connected to it tries to relink itself elsewhere in the tree. If it temporarily fails to do this, it acts as root of a disconnected sub-tree, and load balancing occurs within the sub-tree. Eventually, the flags marking nodes as already tried or dead will be reset, and the computer, which periodically tries to attach itself to the tree, will reconnect to the tree.

Advantageously, the periodic distribution of load information across the links of the tree is used by each computer to determine whether or not the computers to which it is linked are operational, each computer seeking, if the computer

to which it is linked higher in the tree is not oprational, to generate a new link with a computer having lower rank and having the capacity for new downward links and being marked, if one of the computers lower in the tree to which it is linked is not operational, as having capacity to accept new downward links.

Before each distribution of information between computers, entries in the information relating to computers which have the same load and number of links separation from the computer can be randomly permuted, so that the probability that the same entry will appear first in the information stored in different computers is reduced. Also, if an entry corresponding to the computer appears first in its own sorted information a counter can be attached to the entry and initialised to the negative value of the spare load capacity of the computer, the counter being incremented, if that entry is still first, whenever the information is distributed, and if the counter becomes positive that entry is removed from the information. The probability that the same entry will appear first in the information stored in different computers is thereby reduced.

In an advantageous form of the invention the period of the periodic distribution of the information to a computer higher up the tree is related to the rate at which new tasks are created in the computer and/or the rate at which new tasks are created in the computers to which it is logically linked in the tree structure. This has the advantage that if one or more nodes suddenly become highly loaded this information will spread more quickly throughout the network.

A further advantage of the method provided in the present invention is that the case where the overall load in the network increases steadily is handled gracefully. In such a case, when local load vectors are searched for a target computer, fewer candidates will be found and the number of migrations will decrease. When overall load decreases, more candidates will be found in the local load vectors and migrations will resume. In other words the network performance degrades gracefully during network load saturation and does not crash, rather normal operation is resulted when overall load returns to normal.

Viewed from another aspect the invention enables to be provided a computer as claimed in claim 12 capable of operating in a network of similar such computers.

The computer preferably also includes means for accessing a stored list of all the computers in the network; means for selecting a computer from this list as a candidate neighbouring computer for linking in the tree structure; means for sending a message to the selected computer indicating a link request; means for establishing a logical link with the selected computer by updating the identifying means if a positive response from the selected computer is received; means for receiving messages from other computers indicating that a link is requested; means for determining, on receipt of such a link request message, whether or not there is capacity for establishing a downward link and sending a positive response to the sender of the link request message if there is such capacity and updating the identifying means accordingly.

There is also provided a network of such computers.

An embodiment of the invention will now be described, by way of example only, with reference to the accompanying drawing, wherein:

Figures 1a, 1b, 1c and 1d are flow diagrams showing the request and receive phases of the tree generation;

Figures 2a, b and c show stages of the tree generation according to the present invention;

Figures 3a, b and c illustrate an example of tree maintenance in the embodiment of the invention;

Figures 4a, b and c illustrate the handling of clustering of nodes in the embodiment of the invention.

This embodiment of the invention is composed of two main parts. The first part is tree building and maintenance, and the second part is the exchange and maintenance of load balancing information.

TREE BUILDING AND MAINTENANCE

Each computer in the network is assigned a unique rank which is used in building the tree. In the following the term "parent" will be used to refer to a computer of lower rank to which a computer can form an upward link and the term "child" will be used to refer to a computer of higher rank to which a computer can form a downward link.

Each computer has stored in it a configuration file that contains

(i) the maximum number of direct descendants the computer can accept. These are referred to as children slots.

(ii) a list of computers and their respective ranks.

(iii) an ordered list of "favoured parent" nodes, FP. Favoured parents have lower rank than the current computer.

The FP list is used by the computer to choose a parent node. The FP list is optional. It can be empty in which case a parent is chosen at random. The FP list includes a subset of all computers of lower rank.

(iv) the time to wait for parent and children responses.

(v) the time to wait to probe if a node is dead or alive.

A list of candidate parents for a computer CR, which will be referred to as the candidates range of P, limits the range of possible parents. Initially CR of each computer contains all the computers with rank less than the rank of the computer. A node is chosen as a parent only if it is in CR and in FP. If FP is empty or exhausted then only membership of CR is checked.

Each computer performs the following steps to build the tree. There is a request phase in which a computer chooses another computer and requests it to become a parent, and a receive phase in which the computer waits for a reply and responds to requests from other computers.

Request Phase

1. Each computer i picks another computer from its list of favoured parents FP_i. If FP_i is empty or all candidates in FP_i failed, a computer with lesser rank is chosen at random. If the chosen computer j is not in CR_i then the next one in FP_i is chosen or another is randomly picked. If all nodes were tried unsuccessfully then the computer enters the receive phase.

2. i sends the chosen computer j a request_parent message and waits for messages from other computers.

3. If the candidate parent does not respond within a certain period of time, it is marked dead and i reenters the request phase.

Receive Phase

1. If a request_parent message arrives from m then:

(a) If i has a free child slot, it accepts m's request and sends back an ack message

(b) If i has exhausted its allotted number of child slots then:

1) i scans its children for a late child as defined below. If one is found then

(a) the child is marked as dead and removed
(b) this makes a child slot available
(c) i sends m an ack message

If a late child is not found then

2) i finds the child k with highest rank.

3) If rank(k) > rank(m) then:

a) i accepts m as a child in place of k and sends m an ack message.
b) i sends k a disengage message containing rank(m) as a parameter.

4) If rank(k) < rank(m) then:

a) i locates in its list of computers the computer n with the lowest rank satisfying rank(n) > rank(i).
b) i sends m a noack message containing rank(n) as a parameter.

2. If a disengage message with parameter r arrives then:

a) i clears its parent field

b) i removes the computer with ranks outside the range [rank(i)-1,r] from CRi and starts looking for a new parent, ie enters request phase.

3. If ack arrives from candidate parent the computer records the candidate parent as a true parent.

4. If a noack message with parameter r arrives then:

(a) i clears its candidate parent field
(b) the computer executes step 2(b) above.

Figure 1a is a flow diagram showing the request phase of the tree generation and Figures 1b, 1c and 1d are flow diagrams showing the receive phase of the tree generation.

As an example of tree generation, consider 5 computers with ranks 0 to 4 where each can accept two children. Due to random choice of parents the method can reach the stage shown in Figure 2a.

Computer 1 can choose only computer 0 as a parent but computer 0 has already two children. Therefore, the method replaces child 3 of computer 0 with computer 1 and sends computer 3 a disengage message with parameter 1. Thus the situation pictured in Figure 2b is reached.

Node 3 searches for a parent in the range 2 to 1 and picks, say 1 as parent. The resulting tree appears in Figure 2c.

If all nodes are alive and there are no late children then the tree generation terminates with one tree.

The above tree generation may build unbalanced trees or trees with many nodes having only one child. However, it has been found that the load balancing method is insensitive to the topology of the tree.

Tree maintenance is required to detect dead nodes, to add new nodes, or to reconnect rebooted nodes. It is necessary because nodes fail and nodes are sometimes shut down for maintenance. In this embodiment of the invention such cases are handled in the following way.

1. Each computer periodically sends an update_up message to its parent and waits for an update_down message.

2. If no response arrives within a specified period of time the parent is marked dead and the computer looks for a new parent.

3. If an update_up message arrives from a child the node responds with an update_down message.

4. If no update_up message arrives from a child within a specified period of time the child is marked late.

5. If an update_up message arrives from a late child then the late flag for that child is reset and the node responds with an update_down message.

6. When a computer is rebooted or added to the network it looks for a parent, ie it enters the request phase.

7. Each computer flags computers already tried and those marked dead. Periodically these flags are cleared, and CR is reset to contain all computers with lesser rank. Thus rebooted nodes can be probed again.

As an example, consider a tree of 18 nodes labelled a to r as shown in Figure 3a.

If node h fails, then after a predefined period of time, its parent c will mark it as late and ignore it since it did not receive an update_up message. Nodes k and r will start looking for new parents since no update_down message arrived. The network now comprises a number of separate trees as shown in Figure 3b.

k and r look for a new parent in their FP. If their FP is empty they choose a parent at random. The new tree may look like the one in Figure 3c, with node k having node g as its new parent and node r having node c as its new parent.

The tree generation together with the tree maintenance mechanism ensure that the computers in the network generally will be arranged in a single tree structure. To see that this is true in the case of a node failing consider an arbitrary tree in which node k fails or is otherwise inoperational. k's parent will mark it as late and k's children will mark it as dead. Assume m was a child of k. Therefore, m looks for a new parent. There are three possibilities:

1. m finds a node which can accept it as a child. Thus, a single tree is formed.

2. m picks k's parent as a candidate and there are no free slots. In this case the slot of k marked as late is used and k is marked as dead. Again, a single tree is formed.

3. m does not find a node which can accept it as child.
This can happen only if the nodes which would have accepted it (there is at least one, ie m-l) are marked as already tried or as dead. In this case the network temporarily comprises a number of sub-trees rather than a single tree. This state will end when all these flags are reset (step 7 of the maintenance mechanism) and then m will try again and will find a parent.

The above reasoning is applicable for all the children of k.

EXCHANGE AND MAINTENANCE OF INFORMATION

The number of jobs on the run queue is generally accepted as a suitable load metric. A node is considered overloaded if the length of the run queue exceeds a predefined threshold, overload_mark. Since the quantity that is really of interest is how far the node is from its overload mark, a slightly different metric is used in this embodiment of the invention. In this embodiment of the invention load is defined as length_of_run-queue - overload_mark. For example, if $overload\_mark_A = 10$ and $overload\_mark_B = 8$ and the length of run queue of nodes A and B are 5 and 4 respectively, then the $load_A = -5$ and $load_B = -4$. Thus, the load of B is higher than that of A.

In the following this metric will be used for load and whenever the term load is used, length_of_run_queue - overload_mark is meant.

Each computer in the network stores a sorted vector which holds information on other computers in the network. Each entry in the vector contains:

1) the rank of the computer
2) the local load of the computer
3) the distance in terms of the number of edges or links the load information has traversed
4) the last node that propagated this information (last node field)

The length of the load vector, ie the number of entries it contains, may vary from node to node.
Load information is distributed over the network in the following way.

1. Periodically each computer samples its load L, sorts its load vector using the new value of L, and sends the load vector to its parent in an update_up message. The update_up message is also used as the indication that the node is operational for the purposes of tree maintenance.

2. When an update_up message is received by node m from node k,

a The distance of each entry in the received load vector is incremented by 1.

b All entries in the received load vector which originated from receiving node m are deleted.

c All entries in the load vector which originated from the sending node k are deleted.

d The distances of entries (computers) which appear in both the local vector and the received vector are compared:

1. If the distance of local entry is greater or equal to the distance of the received entry then the local entry is deleted.
2. If the distance of the received entry is greater than the distance of the local entry then the received entry is deleted.

e The last node field in the entries of the received vector is set to be the sending node k.

f The received load vector is merged with the local load vector.

3. A parent sends its new load vector to the child who sent the update_up message. This message is an update_down message which again is used as an indication that the parent is operational in the tree maintenance message.

4. When an update_down message arrives, the receiving child processes the received load vector in the same

way as described in step 2.

The update_down messages propagate information received by each node from the nodes to which they are linked in the tree structure. Thus children share information by passing it up to the parent who passes it down to the other children. By propagating load information up and down the tree, information on under-loaded nodes in another sub-tree can reach any node. Due to the fixed size of the load vector, each computer holds information on a sub-set of the computers that are least loaded.

The load information in a load vector is only an approximation of the real load on any particular computer, since by the time it reaches another node the load of the originating computer may have changed.

The 'update interval' is defined as the time interval between successive load distributions; ie update_up messages. This interval significantly influences the results of load balancing. Too large a value renders the load information of the current node in other nodes inaccurate. Migration decisions based on this information result in many rejections due to the inaccuracy of the information. Too small a value; ie a higher update rate, increases the overhead of the method and response time is degraded. The update rate (1/update_interval) should be proportional to the rate of generation of local processes and should change as this rate changes. In addition there is another factor which has to be considered. It has to be ensured that the update rate of the current node does not slow down the propagation of information from its children to its parent and vice versa. This will happen if the rate of the current node is low and that of its children or its parent is high. Therefore, the update rates of the immediate neighbours have also to be considered.

The effective_rate of a node is defined as the result of a rate calculation function based on these two factors.

In this embodiment of the invention the following function is used. Initially the effective rate of each node is set to the birth rate of local processes. For each node p the following quantities are computed:

a) $rate_1 = local\_birth\_rate_p$

Local birth $rate_p$ is computed as an average over a sliding time window.

b) $rate_2 = max\{effective\_rate_n\}/\alpha$

for all immediate neighbours n of p. α is an attenuation factor determined by computer simulation experiments as described below.

The effective_rate of p is given by:

$$effective\_rate_p = max\{rate_1, rate_2\}$$

In computer simulation experiments it was found that values of α in the range 1.4 to 2.0 gave good results for networks with evenly distributed load and networks with regions of high load.

When a computer is overloaded it searches its local load vector for the first entry indicating a computer that can accept extra load, and transfers one or more tasks directly to it. The load entry of the target computer in the load vector is updated and the load vector is re-sorted. If the target computer cannot for whatever reason accept extra load, the sender is notified. In this case the sender deletes that entry from the load vector, retrieves the next entry and use it as a new target. If no node can accept extra load then no migration is initiated.

Thus, the location policy; ie, finding a target node, depends on the sort criterion used to sort the load vector. Since load information is propagated in hops over the tree, the farther away a node is, the less accurate is its information. Therefore, in this embodiment of the evaluation the load vector is sorted according to load and distance. Entries are first sorted according to their load. Entries with equal load are then sorted in ascending order of distance.

A node which is lightly loaded for a long time may appear as the first entry in the load vector of many nodes. This may cause many nodes to migrate processes to that node, thus flooding it and creating a bottleneck. Two approaches can be used to avoid this problem.

An equivalence set of nodes can be defined in the load vector as those nodes with the same load and distance. For example all nodes with load -1 and distance 2 are in the the same equivalence set. In this case, the sort criterion ensures that all the nodes in the same equivalence set are grouped together in the vector. Before a load vector is distributed (by update_up or update_down message) the first equivalence set in the vector undergoes permutation. This reduces the probability that the same node will appear in many load vectors.

In addition, whenever a lightly-loaded node appears first in its own load vector for the first time, a counter is attached to the entry and initialised to be equal to the load of the lightly loaded node which is always less than zero. Whenever the vector is distributed, if that entry is still first, then the counter is incremented. If the counter becomes positive that entry is removed from the vector. Thus the number of nodes which are aware of the lightly loaded node is limited and the probability of flooding reduced.

Initiation of process migration depends on the frequency the local load is checked and on the value of the local load. The decision when to check a node's load status depends on the chosen policy. There are a number of possible policies, including the following

1. Periodic — the local load is tested after sending update_up message. ie, the local load sampling is synchronised with the update_up message.

2. Event driven — The local load is tested whenever a local process is added to the run queue.

The decision when to declare a computer overloaded, ie where to place the overload mark, influences the performance of the method. Placing the overload mark too low may cause too many migrations without improving response time. Placing it too high may degrade response time. The overload mark value needs to be optimised for any particular system and the value may be different for computers in the network having different load capacities or for different job type profiles. The optimum value or values can be established in any given situation by the use, for example, of appropriate computer simulation experiments or by a process of trial and error.

A controlled clustering of nodes can be achieved by an appropriate assignment of rank in the configuration files of each node.

For example, if there are eleven computers a to k and it is required to form clusters of four and seven computers as follows,

a	b	c	d		e	f	g	h	i	j	k
1st cluster					2nd cluster						

the following steps are performed to rank the nodes.

1. Each cluster is ranked separately. For example

a	b	c	d		e	f	g	h	i	j	k
1	2	0	3		5	6	0	1	2	3	4

2. Each cluster is taken in turn, and rank of each node is increased by the number of nodes in previous clusters. For example if the 2nd cluster is chosen first.

a	b	c	d		e	f	g	h	i	j	k
8	9	7	10		5	6	0	1	2	3	4

Cluster configuration is achieved by assigning different ranking files to the computers. Each computer except the one with lowest rank in a cluster, stores a ranking list that contains ranks of nodes in its cluster and not those outside the cluster. The computer with the lowest rank stores the ranking list with nodes in other clusters.

Nodes in a cluster will therefore connect to each other since they are not aware of other nodes. Only the node with the lowest rank will connect to a node in another cluster, because it cannot connect to another node in the cluster. Figure 4a shows a sample tree formed from the nodes in the above example.

If the maximum number of children for c is three, and the favoured parent lists of a, b and d contain only c, the tree described in Figure 4b may result.

Placing j in the favoured parent list of c may generate the tree described in Figure 4c.

Thus clustering is achieved in the embodiment of the invention by judicious choice of ranking files and favoured parents lists for each node.

Whilst the invention has been described in terms of an embodiment using a sender-initiated load balancing policy in which an overloaded node initiates the transfer of tasks, it will be clear to those skilled in the art that the invention could equally well be applied to a receiver-initiated scheme in which a lightly loaded computer initiates transfer of tasks from a heavily loaded computer to itself.

There has been described a method of operating a computer in a network and operating a network of computers using an improved load balancing technique which is scalable, fault tolerant, flexible and supports clustering thus making it suitable for use in networks having very large numbers of computers.

Claims

1. A method of operating a computer in a network of computers, the method comprising:

   generating logical links between the computer and other computers in the network so that a tree structure is formed, the computer being logically linked to one computer higher up the tree and a number of computers lower down the tree; and **characterised by**

   maintaining in the computer stored information regarding the current load on the computer and the load on at least some of the other computers in the network by causing the computer periodically to distribute the information to the computers to which it is logically linked, to receive from said computers similar such information and to update its own information in accordance therewith, so that the information can be used to determine computers in the network that can accept extra load;

   wherein the information stored in the computer contains a number of entries, each entry containing information regarding:

   (i) the load on one of the computers in the network,

   (ii) the number of links in the tree separating that other computer from the computer, and

   (iii) the one of the computers, which are linked to the computer, from which the entry was last received;

   and wherein when the computer receives the similar information from a computer to which it is linked the following steps are performed:

   (a) the number of links value in each entry of the received information is incremented by one;

   (b) entries in the received information which originated from the computer are deleted;

   (c) entries in the information already stored in the computer which were received from the other computer are deleted;

   (d) the received information is merged with the information already stored in the computer;

   (e) the merged information is sorted in ascending order of load, entries with equal load being sorted in ascending order of number of links separation from the computer.

2. A method as claimed in claim 1 comprising the step of, when the computer is overloaded, using the information to determine a computer that can accept extra load and transferring at least one task to that computer.

3. A method as claimed in claim 1 or claim 2 comprising, if the computer higher up the tree, to which the computer is logically linked fails or is otherwise inoperative, generating a new logical link to another computer in the network which has capacity for accepting new downward links.

4. A method as claimed in any preceding claim wherein the periodic distribution of load information is used by the computer to determine whether or not the computers to which it is linked in the tree, are operational, the computer being marked, if one of the computers lower in the tree to which it is linked is not operational, as having capacity to accept new downward links.

5. A method as claimed in claim 1 wherein before each distribution of information entries in the information relating to computers which have the same load and number of links separation from the computer are randomly permuted.

6. A method as claimed in claim 1 or 5 wherein if an entry corresponding to the computer appears first in its own sorted information a counter is attached to the entry and initialised to negative value of the spare load capacity of the computer, the counter being incremented, if that entry is still first, whenever the information is distributed, and if the counter becomes positive that entry is removed from the information, whereby the probability that the same entry will appear first in the information stored In different computers is reduced.

7. A method as claimed in any preceding claim wherein the period of the periodic distribution of the information to a computer higher up the tree depends on the rate at which new tasks are created in the computer

8. A method as claimed in claim 9 wherein the period of the periodic distribution depends on the rate at which new tasks are created in the computers to which the computer is logically linked in the tree structure.

9. A method as claimed in any preceding claim wherein the distribution of the information to a computer lower in the tree takes place in response to receipt by the computer of the similar information from the computer lower in the tree.

10. A method of operating a network of computers, comprising operating each computer using a method as claimed in any preceding claim.

11. A method as claimed in claim 10 wherein the generation of the logical links is achieved by assigning a rank to each computer, no two computers being assigned the same rank, each computer being linked to one computer of lower rank and a number of computers of higher rank to form the tree structure.

12. A computer capable of operating in a network of similar such computers, the computer comprising:

means identifying computers in the network to which the computer is logically linked in a tree structure;

means for storing information regarding the load on the computer and at least some of the other computers in the network;

means for sending said information to the computers to which the computer is logically linked;

means for receiving similar information from said computers to wnich the computer is logically linked and updating the stored information in accordance therewith;

means for selecting one of the other computers in the network having spare load capacity from said information and transferring tasks to the selected computer;

characterised in that the information stored in the computer contains a number of entries, each entry containing information regarding:

(i) the load on one of the computers in the network,

(ii) the number of links in the tree separating that other computer from the computer, and

(iii) the one of the computers, which are linked to the computer, from which the entry was last received;

and wherein, the computer further comprises means, operable when the computer receives the similar information from one of the other computers to which it is linked, for:

(a) incrementing the number of links value in each entry of the received information by one;

(b) deleting entries in the received information which originated from the other computer;

(c) deleting entries in the information already stored in the computer which were received from the other computer;

(d) merging the received information with the information already stored in the computer;

(e) sorting the merged information in ascending order of load, entries with equal load being sorted in ascending order of number of iinks separation from the computer.

13. A computer as claimed in claim 12 including:

means for accessing a stored list of all the computers in the network;

means for selecting a computer from this list as a candidate neighbouring computer for linking in the tree structure;

means for sending a message to the selected computer indicating a link request;

means for establishing a logical link with the selected computer by updating the identifying means if a positive response from the selected computer is received;

means from receiving messages from other computers indicating that a link is requested;

means for determining, on receipt of such a link request message, whether or not there is capacity for establishing a downward link and sending a positive response to the sender of the link request message if there is such capacity and updating the identifying means accordingly.

14. A network of computers comprising a plurality of computers as claimed in claim 12 or claim 13.

Patentansprüche

1. Verfahren zum Betrieb eines Rechners in einem Rechnernetz, wobei das Verfahren beinhaltet:

Generieren logischer Verbindungsglieder zwischen dem Rechner und anderen Rechnern im Netz, so daß sich eine Baumstruktur bildet, wobei der Rechner logisch mit einem im Baum höher stehenden Rechner und mit mehreren im Baum tiefer stehenden Rechnern verbunden ist; und gekennzeichnet durch

Unterhalten von im Rechner gespeicherten Informationen betreffend die augenblickliche Auslastung des Rechners und die Auslastung wenigstens einiger der anderen Rechner im Netz durch Bewirken, daß der Rechner periodisch diese Informationen an die Rechner weitergibt, mit denen er logisch verbunden ist, um ähnliche solcher Informationen von diesen Rechnern zu erhalten und seine eigenen Informationen entsprechend zu aktualisieren, so daß die Informationen benutzt werden können, um Rechner im Netz zu bestimmen, die Extralasten aufnehmen können;

wobei die im Rechner gespeicherten Informationen eine Anzahl Einträge enthalten, und jeder Eintrag Informationen über folgende Punkte enthält:

(i) Die Belastung eines der Rechner im Netz,

(ii) die Anzahl der Verbindungsglieder im Baum, die diesen anderen Rechner vom Rechner trennen, und

(iii) denjenigen der Rechner, die mit dem Rechner verbunden sind, von dem der Eintrag zuletzt erhalten wurde;

und in dem bei Eingang ähnlicher Informationen von einem Rechner, mit dem er verbunden ist, die folgenden Schritte ausgeführt werden:

(a) Die Höhe der Anzahl der Verbindungsglieder in jedem Eintrag der erhaltenen Informationen wird um eins erhöht;

(b) Einträge in den erhaltenen Informationen, die von dem Rechner ausgingen, werden gelöscht;

(c) Einträge in den im Rechner bereits gespeicherten Informationen, die von dem anderen Rechner ein-

gingen, werden gelöscht;

(d) die eingegangenen Informationen werden mit den bereits in Rechner gespeicherten Informationen verbunden;

(e) die verbundenen Informationen werden in aufsteigender Reihenfolge der Auslastung sortiert, Einträge gleicher Auslastung werden dabei in aufsteigender Reihenfolge der Verbindungsglieder, um die er vom Rechner getrennt ist, sortiert.

2. Ein Verfahren gemäß Anspruch 1, das den folgenden Schritt enthält: Wenn der Rechner überlastet ist, Benutzen der Information, um eine Rechner zu bestimmen, der eine Extralast übernehmen kann, und Übertragen wenigstens einer Aufgabe auf diesen Rechner.

3. Ein Verfahren gemäß Anspruch 1 oder Anspruch 2, das den folgenden Schritt enthält: Wenn der höher im Baum sitzende Rechner, mit dem der Rechner logisch verbunden ist, ausfällt oder sonstwie nicht betriebsbereit ist, Generieren eines neuen logischen Verbindungsglieds mit einem anderen Rechner im Netz, der noch Kapazität zum Übernehmen neuer, nach unten gerichteter Verbindungsglieder hat.

4. Ein Verfahren gemäß einem beliebigen der vorstehenden Ansprüche, in dem die periodische Verteilung von Lastinformationen vom Rechner benutzt wird, um festzustellen, ob die Rechner, mit denen er im Baum verbunden ist, betriebsbereit sind oder nicht, wobei der Rechner, wenn einer der weiter unten im Baum sitzenden Rechner, mit denen er verbunden ist, nicht betriebsbereit ist, als Kapazität zur Annahme neuer nach unten gerichteter Verbindungen aufweisend markiert wird.

5. Ein Verfahren gemäß Anspruch 1, in dem vor jeder Verteilung von Informationseinträgen in die Informationen bezüglich der Rechner, die die gleiche Auslastung und Trennung um die gleiche Anzahl von Verbindungsgliedern von dem Rechner aufweisen, nach Zufallsverteilung permutiert werden.

6. Ein Verfahren gemäß Anspruch 1 oder 5, in dem, wenn sich ein Eintrag auf den Rechner selbst bezieht, dieser als erster in seinen eigenen sortierten Informationen erscheint, ein Zähler an den Eintrag angeschlossen und auf den negativen Wert der Reservelastkapazität des Rechners initialisiert wird, wobei der Zähler jedesmal inkrementiert wird, wenn dieser Eintrag immer noch der erste ist, wenn die Information verteilt wird, und sobald der Zähler positiv wird, dieser Eintrag aus der Information entfernt wird, wobei sich die Wahrscheinlichkeit, daß der gleiche Eintrag als erster in den in verschiedenen Rechnern gespeicherten Informationen erscheint, reduziert.

7. Ein Verfahren gemäß einem beliebigen der vorstehenden Ansprüche, in dem die Periode der periodischen Verteilung der Information an einen höher im Baum sitzenden Rechner von der Rate abhängt, mit der neue Aufgaben im Rechner geschaffen werden.

8. Ein Verfahren gemäß Anspruch 7, in dem die Periode der periodischen Verteilung von der Rate abhängt, mit der neue Aufgaben in Rechnern geschaffen werden, mit denen der Rechner in der Baumstruktur logisch verbunden ist.

9. Ein Verfahren gemäß einem beliebigen der vorstehenden Ansprüche, in dem die Verteilung der Informationen durch den Rechner an einen weiter unten im Baum sitzenden Rechner als Antwort auf den Eingang der ähnlichen Information vom weiter unten im Baum sitzenden Rechner erfolgt.

10. Ein Verfahren zum Betreiben eines Rechnernetzes, beinhaltend das Betreiben jedes Rechners unter Verwendung eines Verfahrens, wie es in beliebigen der vorstehenden Ansprüche beansprucht ist.

11. Ein Verfahren gemäß Anspruch 10, in dem das Generieren der logischen Verbindungsglieder erzielt wird durch Zuweisung eines Rangs an jeden Rechner, wobei nicht zwei Rechner den gleichen Rang zugeteilt bekommen, jeder Rechner logisch mit einem Rechner eines niedrigeren Rangs und mit einer Anzahl Rechner höheren Rangs verbunden ist, um so eine Baumstruktur zu bilden.

12. Ein Rechner, der in der Lage ist, in einem Netz aus ähnlichen solchen Rechnern zu arbeiten, wobei der Rechner beinhaltet:

Mittel zum Identifizieren von Rechnern im Netz, mit denen der Rechner in einer Baumstruktur logisch verbun-

den ist;

Mittel zum Abspeichern der Informationen betreffend die Auslastung des Rechners und wenigstens einiger der anderen Rechner im Netz;

Mittel zum Senden der Informationen an die Rechner, mit denen der Rechner logisch verbunden ist;

Mittel zum Empfangen ähnlicher Informationen von den Rechnern, mit denen der Rechner logisch verbunden ist, und Aktualisieren der abgespeicherten Informationen in Übereinstimmung damit;

Mittel zum Anwählen eines der anderen Rechner mit freien Auslastungskapazitäten im Netz aus diesen Informationen und Übertragen von Aufgaben auf diesen angewählten Rechner;

dadurch gekennzeichnet, daß die im Rechner gespeicherten Informationen eine Anzahl Einträge enthalten, und jeder Eintrag Informationen über folgende Punkte enthält:

(i) Die Belastung eines der Rechner im Netz;

(ii) die Anzahl der Verbindungsglieder im Baum, die diesen anderen Rechner vom Rechner trennen, und

(iii) denjenigen der Rechner, die mit dem Rechner verbunden sind, von dem der Eintrag zuletzt erhalten wurde;

und in dem der Rechner ferner Mittel beinhaltet, die betreibbar sind, wenn der Rechner diese ähnlichen Informationen von einem der anderen Rechner erhält, mit dem er verbunden ist, um die folgenden Maßnahmen zu ergreifen:

(a) Inkrementieren des Werts der Anzahl der Verbindungsglieder in jedem Eintrag der erhaltenen Informationen um eins;

(b) Löschen der Einträge in den erhaltenen Informationen, die von dem anderen Rechner stammen;

(c) Löschen der Einträge in den im Rechner bereits gespeicherten Informationen, die von dem anderen Rechner eingegangen waren;

(d) Verbindung der eingegangen Informationen mit den bereits in Rechner gespeicherten Informationen;

(e) Sortieren der verbundenen Informationen in aufsteigender Reihenfolge der Auslastung, wobei Einträge gleicher Auslastung in aufsteigender Reihenfolge der Anzahl der Verbindungsglieder, um die er vom Rechner getrennt ist, sortiert werden.

13. Ein Rechner gemäß Anspruch 12, beinhaltend:

Mittel, um auf eine gespeicherte Liste aller Rechner im Netz zuzugreifen;

Mittel zum Wählen eines Rechners aus der Liste als benachbarten Kandidaten-Rechner zum Verbinden in der Baumstruktur;

Mittel zum Senden einer Meldung an den gewählten Rechner mit Anforderung der Verbindung;

Mittel zum Errichten einer logischen Verbindung mit dem gewählten Rechner durch Aktualisieren der Identifizierungsmittel, wenn eine positive Antwort vom angewählten Rechner eingeht;

Mittel zum Empfangen von Meldungen anderer Rechner, die anzeigen, daß eine Verbindung angefordert wird;

Mittel zum Bestimmen bei Eingang einer solchen verbindungsanfordernden Meldung, ob es eine Kapazität zum Aufbau einer nach unten gerichteten Verbindung gibt oder nicht, und Senden einer positiven Meldung an den Sender der verbindungsanfordernden Meldung, wenn eine solche Kapazität vorhanden ist, und entsprechende Aktualisierung der Identifizierungsmittel.

14. Ein Rechnernetz, enthaltend eine Vielzahl von Rechnern gemäß den Ansprüchen 12 oder 13.

Revendications

1. Procédé pour exploiter un ordinateur dans un réseau d'ordinateurs, le procédé comprenant les phases qui consistent à:

   créer des liaisons logiques entre l'ordinateur et d'autres ordinateurs du réseau de manière à former une structure arborescente, l'ordinateur étant relié logiquement à un ordinateur situé plus haut dans l'arbre et à plusieurs ordinateurs situés plus bas dans l'arbre ; et caractérisé par le fait de

   conserver dans l'ordinateur un enregistrement de l'information relative à la chargeen mémoire courante de l'ordinateur et de la charge en mémoire d'au moins certains des autres ordinateurs du réseau, en obligeant l'ordinateur à répartir périodiquement l'information entre les ordinateurs auxquels il est relié logiquement, pour recevoir des dits ordinateurs des informations similaires et pour mettre à jour sa propre information conformément à ce qu'il a reçu, de sorte que l'information peut être utilisée pour déterminer quels sont les ordinateurs du réseau qui peuvent accepter une charge supplémentaire ;

   où l'information enregistrée dans l'ordinateur comprend de nombreuses entrées, chaque entrée contenant l'information relative à :

   (i) la charge en mémoire d'un des ordinateurs du réseau,
   (ii) le nombre de liaisons d'arborescence séparant cet autre ordinateur de l'ordinateur, et
   (iii) celui, parmi les ordinateurs qui sont reliés à l'ordinateur, depuis lequel a été envoyée l'entrée reçue en dernier;

   et où, quand l'ordinateur reçoit l'information similaire envoyée par un ordinateur auquel il est relié, les phases suivantes sont exécutées :

   (a) la valeur du nombre de liaisons dans chaque entrée de l'information reçue est augmentée d'un incrément ;
   (b) dans l'information reçue les entrées qui provenaient à l'origine de l'ordinateur sont effacées;
   (c) les entrées de l'information déjà enregistrée dans l'ordinateur qui ont été reçues en provenance de l'autre ordinateur sont effacées ;
   (d) l'information reçue est fusionnée à l'information déjà enregistrée dans l'ordinateur ;
   (e) l'information fusionnée est triée par ordre croissant d'importance de la charge, les entrées ayant une même charge étant triées par ordre croissant du nombre de liaisons les séparant de l'ordinateur.

2. Procédé selon la revendication 1, comprenant la phase qui consiste, quand la mémoire de l'ordinateur est trop chargée, à utiliser l'information pour déterminer quel ordinateur peut accepter une charge supplémentaire et à transférer au moins une tâche vers cet ordinateur.

3. Procédé selon la revendication 1 ou la revendication 2 comprenant, si l'ordinateur, situé plus haut dans l'arbre, auquel l'ordinateur est relié logiquement est défaillant ou incapable de fonctionner pour une quelconque raison, la création d'une nouvelle liaison logique vers un autre ordinateur du réseau qui peut accepter de nouvelles liaisons vers le bas.

4. Procédé selon l'une quelconque des revendications précédentes, où la répartition périodique de l'information relative à la charge en mémoire est utilisée par l'ordinateur pour déterminer si les ordinateurs auxquels il est relié dans l'arborescence sont opérationnels ou pas, l'ordinateur étant signalé comme pouvant accepter de nouvelles liaisons vers le bas si l'un des ordinateurs auxquels il est relié et qui se trouve plus bas dans l'arborescence, n'est pas opérationnel.

5. Procédé selon la revendication 1 où, avant chaque répartition de l'information, les entrées relatives aux ordinateurs qui ont la même charge et le même nombre de liaisons de séparation avec l'ordinateur sont permutées au hasard.

6. Procédé selon la revendication 1 ou 5 où, si une entrée correspondant à l'ordinateur apparaît en premier dans sa

propre information triée, un compteur est associé à l'entrée et initialisé avec la valeur négative de la capacité de charge restante de l'ordinateur, le compteur augmentant d'un incrément, si cette entrée est toujours la première, quand l'information est répartie et, si le compteur devient positif cette entrée est retirée de l'information, grâce à quoi la probabilité pour que la même entrée apparaisse en premier dans l'information enregistrée dans différents ordinateurs se trouve réduite.

7. Procédé selon l'une quelconque des revendications précédentes, où la périodicité de la répartition périodique de l'information vers un ordinateur placé plus haut dans l'arborescence dépend de la vitesse à laquelle de nouvelles tâches sont créées dans l'ordinateur.

8. Procédé selon la revendication 9 où la périodicité de la répartition périodique dépend de la vitesse à laquelle de nouvelles tâches sont créées dans les ordinateurs auxquels l'ordinateur est relié logiquement dans la structure arborescente.

9. Procédé selon l'une quelconque des revendications précédentes où la répartition de l'information vers un ordinateur placé plus bas dans l'arbre est exécutée en réponse à la réception, par l'ordinateur, de l'information similaire émanant de l'ordinateur placé plus bas dans l'arbre.

10. Procédé pour exploiter un réseau d'ordinateurs, où chaque ordinateur est exploité en utilisant un procédé conforme à l'une quelconque des revendications précédentes.

11. Procédé selon la revendication 10, où on obtient la création des liaisons logiques en assignant un rang à chaque ordinateur, deux ordinateurs ne se voyant pas assigner le même rang, chaque ordinateur étant relié à un ordinateur de rang inférieur et à plusieurs ordinateurs de rang supérieur pour former la structure arborescente.

12. Ordinateur pouvant être exploité dans un réseau composé d'ordinateurs similaires, l'ordinateur comprenant :

un moyen identifiant les ordinateurs du réseau auxquels l'ordinateur est relié logiquement dans une structure arborescente ;

un moyen pour enregistrer l'information relative à la charge en mémoire de l'ordinateur et à celle d'au moins certains des autres ordinateurs du réseau ;

un moyen envoyant ladite information aux ordinateurs auxquels l'ordinateur est relié logiquement ;

un moyen pour recevoir l'information similaire envoyée par les dits ordinateurs auxquels l'ordinateur est relié logiquement et pour mettre à jour l'information enregistrée, conformément à ce qui a été reçu ;

un moyen pour sélectionner, à partir de ladite information, un des autres ordinateurs du réseau auquel il reste de la capacité de charge et pour transférer des tâches à l'ordinateur sélectionné ;

caractérisé en ce que l'information enregistrée dans l'ordinateur comprend plusieurs entrées, chaque entrée contenant de l'information relative à :

(i) la charge en mémoire d'un des ordinateurs du réseau,
(ii) le nombre de liaisons, dans l'arbre, séparant cet autre ordinateur de l'ordinateur, et
(iii) parmi ceux qui sont reliés à l'ordinateur, l'ordinateur depuis lequel l'entrée reçue en dernier a été envoyée ;

et où l'ordinateur comprend en outre un moyen qui, quand l'ordinateur reçoit l'information similaire envoyée par un ordinateur auquel il est relié, sert à :

(a) ajouter un incrément à la valeur du nombre de liaisons dans chaque entrée de l'information reçue ;
(b) effacer dans l'information reçue les entrées qui provenaient à l'origine de l'autre ordinateur ;
(c) effacer les entrées de l'information déjà enregsitrée dans l'ordinateur qui ont été reçues d'un autre ordinateur ;
(d) fusionner l'information reçue avec l'information déjà enregistrée dans l'ordinateur;
(e) trier par ordre croissant d'importance de la charge l'information fusionnée, les entrées ayant une même charge étant classées par ordre croissant du nombre de liaisons les séparant de l'ordinateur.

13. Ordinateur selon la revendication 12 comprenant :

un moyen pour accéder à l'enregistrement d'une liste de tous les ordinateurs du réseau ;

un moyen pour sélectionner, à partir de cette liste, un ordinateur voisin comme candidat possible pour la création d'une liaison dans la structure arborescente ;

un moyen pour envoyer à l'ordinateur sélectionné un message indiquant une requête de liaison ;

un moyen pour établir une liaison logique avec l'ordinateur sélectionné en mettant à jour le moyen d'identification si une réponse positive a été reçue de l'ordinateur sélectionné ;

un moyen pour recevoir les messages des autres ordinateurs indiquant qu'il y a une requête pour une liaison ;

un moyen pour déterminer, sur réception d'un message de requête de liaison, s'il est possible ou non d'établir une liaison vers le bas et pour envoyer une réponse positive à l'émetteur du message de requête de liaison si c'est possible et pour mettre à jour en conséquence le moyen d'identification.

14. Réseau d'ordinateur comprenant une pluralité d'ordinateurs tels que revendiqués dans la revendication 12 ou 13.

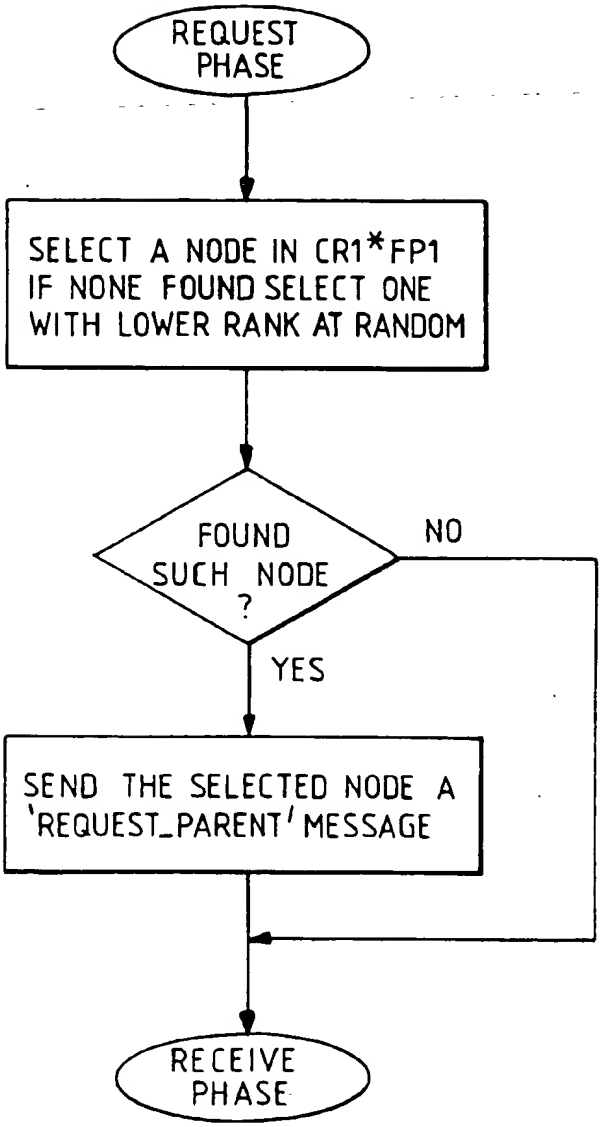
REQUEST PHASE
SELECT A NODE IN CR1*FP1 IF NONE FOUND SELECT ONE WITH LOWER RANK AT RANDOM
FOUND SUCH NODE ?
NO
YES
SEND THE SELECTED NODE A 'REQUEST_PARENT' MESSAGE
RECEIVE PHASE

FIG.1a

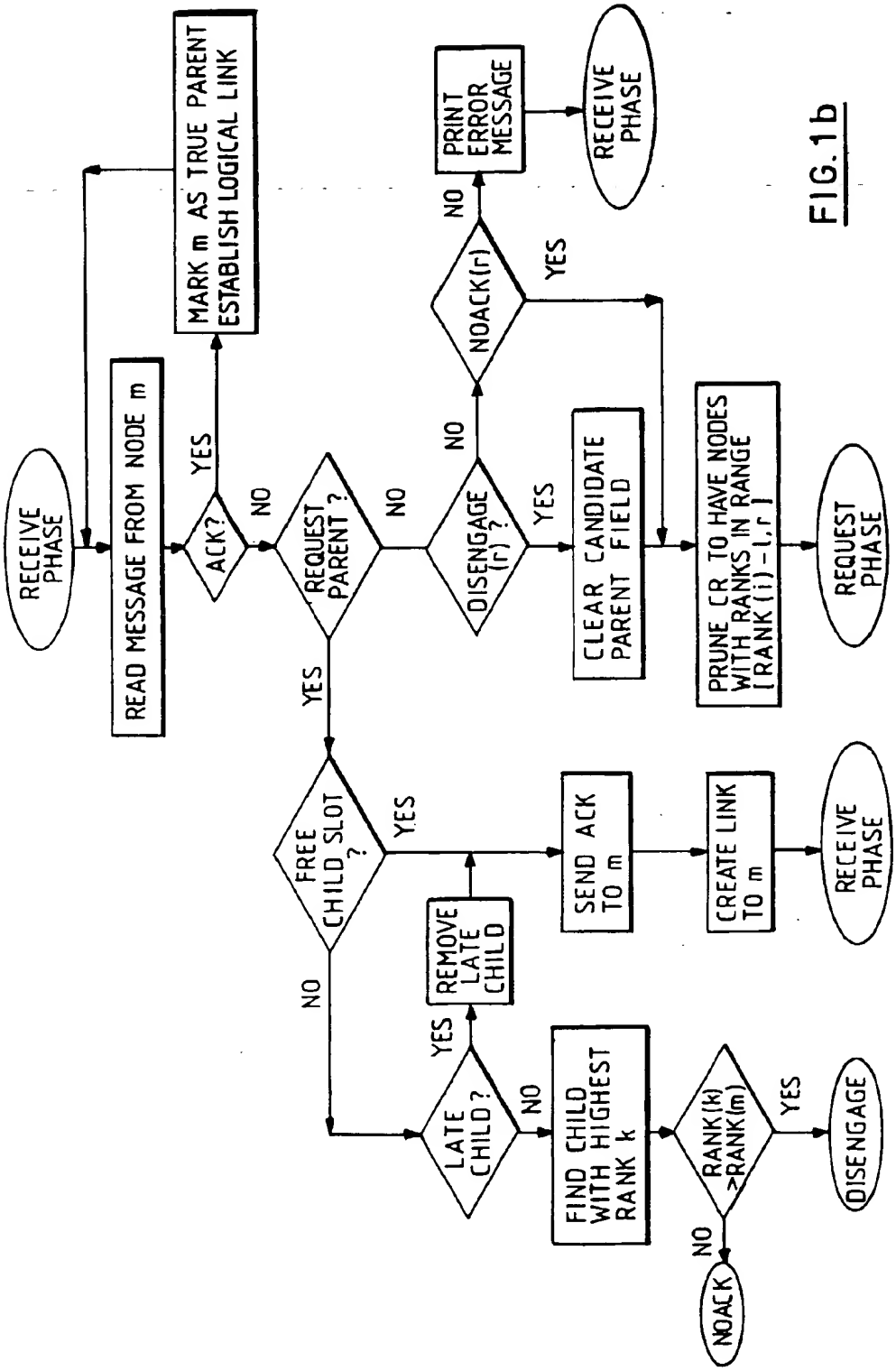
RECEIVE PHASE
READ MESSAGE FROM NODE m
ACK?
YES
NO
MARK m AS TRUE PARENT
ESTABLISH LOGICAL LINK
REQUEST PARENT ?
NO
YES
DISENGAGE (r) ?
NO
YES
NOACK(r)
NO
YES
PRINT ERROR MESSAGE
RECEIVE PHASE
CLEAR CANDIDATE PARENT FIELD
PRUNE CR TO HAVE NODES WITH RANKS IN RANGE [RANK(i)-1,r]
REQUEST PHASE
FREE CHILD SLOT ?
YES
NO
SEND ACK TO m
CREATE LINK TO m
RECEIVE PHASE
REMOVE LATE CHILD
LATE CHILD ?
YES
NO
FIND CHILD WITH HIGHEST RANK k
RANK(k) >RANK(m)
YES
NO
DISENGAGE
NOACK

FIG.1b

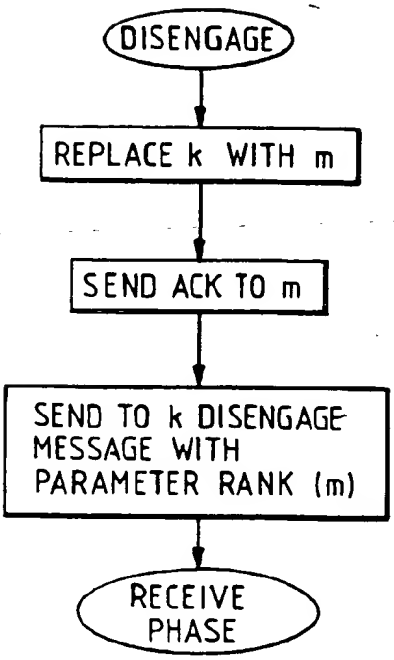
DISENGAGE
REPLACE k WITH m
SEND ACK TO m
SEND TO k DISENGAGE MESSAGE WITH PARAMETER RANK (m)
RECEIVE PHASE

FIG. 1c

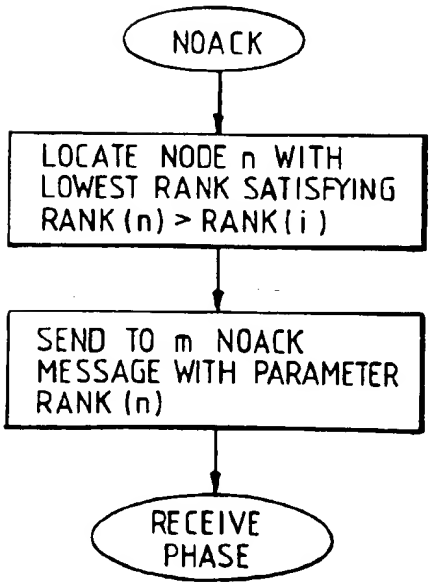
NOACK
LOCATE NODE n WITH LOWEST RANK SATISFYING RANK(n) > RANK(i)
SEND TO m NOACK MESSAGE WITH PARAMETER RANK (n)
RECEIVE PHASE

FIG. 1d

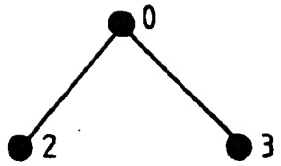

FIG. 2a

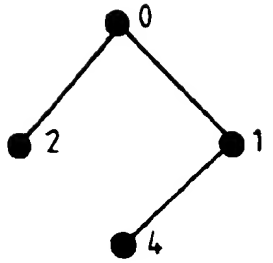

FIG. 2b

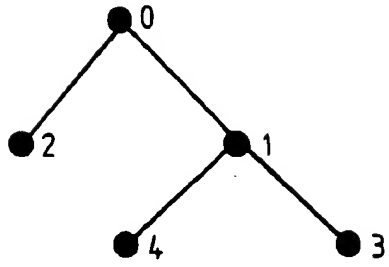

FIG. 2c

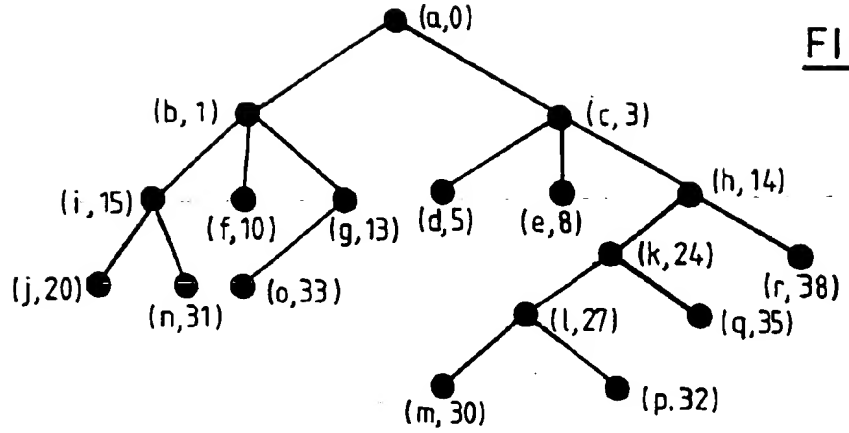

FIG. 3a

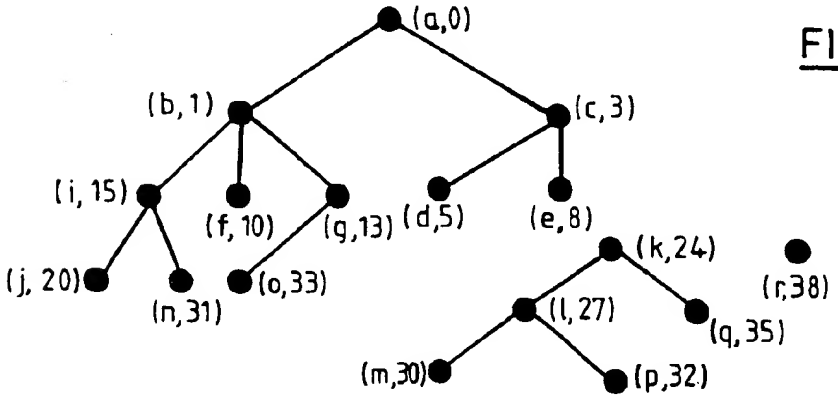

FIG. 3b

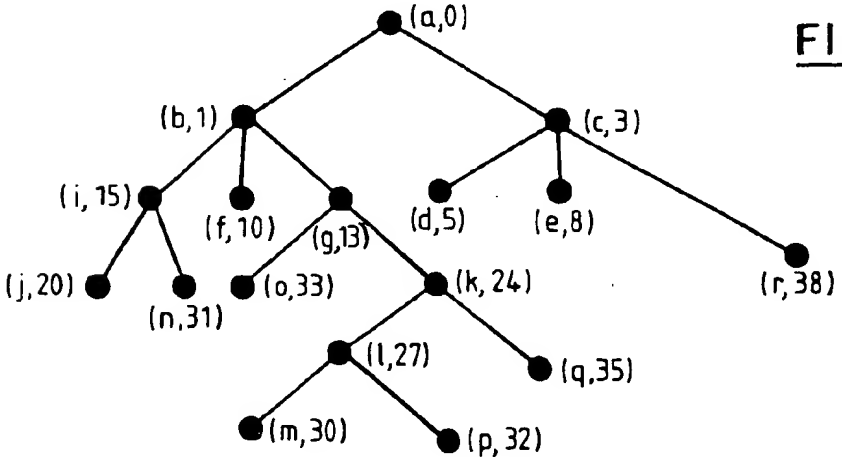

FIG. 3c

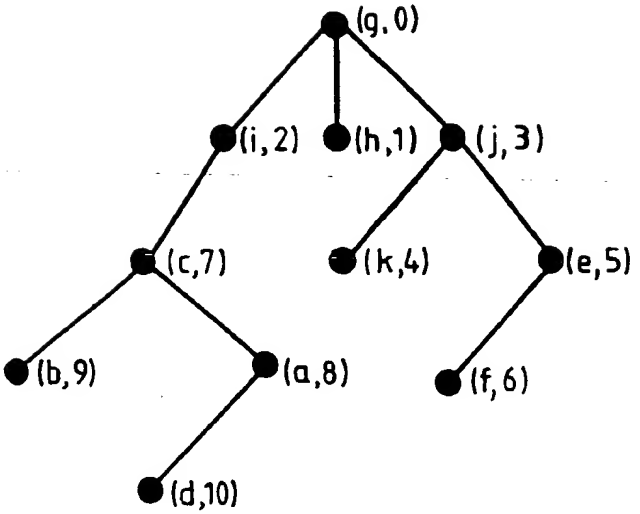

FIG. 4a

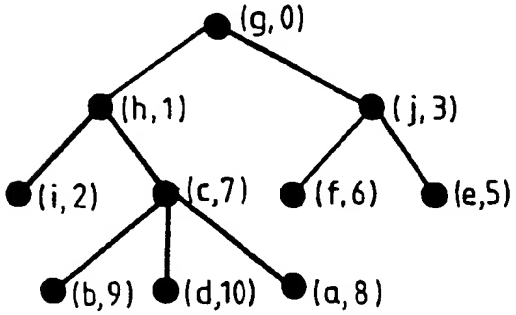

FIG. 4b

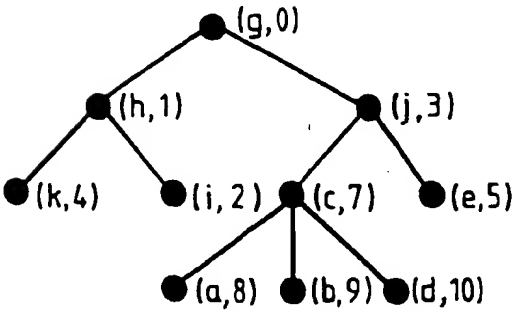

FIG. 4c